A DISTRIBUTED KERNEL FOR SIMULATION OF THE VHSIC HARDWARE DESCRIPTION LANGUAGE

THESIS

Michael Chris Proicou
Captain, USAF

AFIT/GCS/ENG/89D-14

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/89D-14

A DISTRIBUTED KERNEL FOR SIMULATION OF THE VHSIC HARDWARE DESCRIPTION LANGUAGE

THESIS

Michael Chris Proicou
Captain, USAF

AFIT/GCS/ENG/89D-14

AFIT/GCS/ENG/89D-14

A DISTRIBUTED KERNEL FOR SIMULATION OF THE VHSIC HARDWARE DESCRIPTION LANGUAGE

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Science)

Michael Chris Proicou, B.S.

Captain, USAF

December, 1989

Acknowledgments

I'd like to thank my thesis committee for their input and guidance in completing this study. Dr. Thomas Hartrum has been especially helpful in keeping me from straying too far from the straight-and-narrow.

Michael Chris Proicou

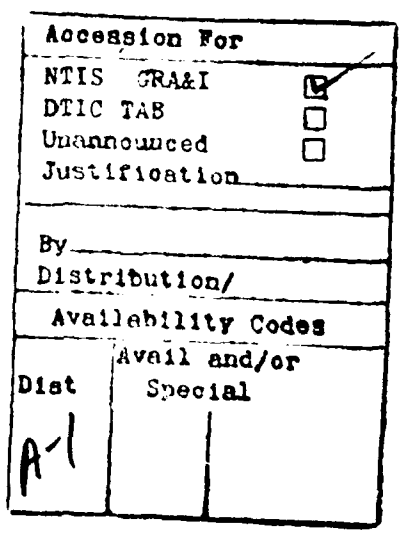

Table of Contents

List of Figures

List of Tables

Abstract

The Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) program was started in 1983 to standardize the tools used in Air Force applications to design, test, and document large-scale digital electronic systems. The simulation of large-scale electronic designs using VHDL is a very daunting task, and is rapidly outgrowing the resources available on single processor machines. AFIT has been developing a set of tools for using VHDL, including an analyzer and simulator for single processor machines.

This thesis develops a technique for simulating large-scale VHDL models on a distributed computer composed of many individual processing units. The distributed system consists of a scalable kernel which can support a large simulation composed of many concurrent VHDL processes. The kernel provides model-independent support functions that handle signal propagation and process activation in a distributed environment. The synchronization between individual logical processes in the kernel is handled using the Chandy-Misra null message protocol. The distributed simulation kernel extends the AFIT tools to include parallel simulation for large-scale VHDL models.

The distributed kernel has been implemented using the University of Virginia's Spectrum portable simulation testbed and runs on the eight-processor Intel iPSC/2 Hybercube computer at AFIT. Several test models have been simulated, including ripple-carry and carry look-ahead adders, and a counter. These cases cover sequential and combinational circuitry, and exercise several features of VHDL. The VHDL statements used to write the test cases are: concurrent signal assignment statements, process statements, and process statements with wait-conditions. The performance of this distributed kernel has been disappointing. The test cases take the same execution time when the number of processors is increased. This is because the number of messages processed is exactly the same on two processors as it is on eight processors.

The characteristics of a VHDL simulation can vary greatly between different circuit models. This variation affects the performance provided by any one simulation kernel design. If the model and kernel are mismatched, poor performance will result. Efficient dis-

tributed simulation of VHDL will require methods of selecting synchronization techniques that match the characteristics of the circuit under test.

A DISTRIBUTED KERNEL FOR SIMULATION OF THE VHSIC HARDWARE DESCRIPTION LANGUAGE

I. Introduction

The size and complexity of digital electronic designs has increased dramatically over the past 20 years. The use of very large scale integrated circuitry threatens to overwhelm the capabilities of present-day design and documentation tools. The Air Force has a large investment in electronic hardware, and the development costs of new hardware are growing as the hardware becomes more complex. The Department of Defense started the Very-High Speed Integrated Circuit (VHSIC) program in 1979 to develop and promote the use of high-density integrated circuits in military systems (10). One of the goals of the VHSIC program is to standardize the methods of describing and documenting hardware designs. The VHSIC Hardware Description Language (VHDL) program was started in 1983 to standardize the tools needed to efficiently design, test, and document large-scale digital electronics. The initial implementations of VHDL were developed by Intermetrics, Inc. under contract in 1983. After several revisions, the IEEE Standard VHDL Language Reference Manual was published in 1987. VHDL has become important enough in recent years that the Department of Defense Advanced Research Projects Agency (DARPA) has sponsored the QUEST project, whose goal is a thousand-fold speed-up in VHDL simulation by 1991. The three partners in QUEST are the University of Cincinnati, University of Virgina, and AFIT.

One advantage of using VHDL occurs when the electrical and physical properties of a circuit are documented in VHDL as the design progresses. The VHDL description of the circuit can be automatically converted to circuit diagrams and layouts by computer-aided-design tools. This will ensure that the documented specification matches the actual hardware. Thus, VHDL can serve as partial documentation of the circuit's function and structure (10:13).

Another benefit of VHDL occurs in the computer simulation of proposed designs. The simulation of circuits and chips allows the designer to test alternative circuits before chips are fabricated or prototypes built. The simulation environment allows a chip or circuit to be tested under conditions and signals that are expected to occur in the actual system. Since very large scale integration (VLSI) designs are so complicated, computer simulation allows more tests to be performed than could be done using a physical test station. This helps increase the reliability of the circuits under development.

AFIT has supported the VHDL program through research leading to development of an analyzer and simulator for UNIX-based computers (21, 9). These tools complement the commercially available analyzers and simulators for other computer systems, and are geared towards educational institutions.

Simulation testing of VLSI-class components introduces complexities of its own. The simulation of a device with 100,000 transistors that has events occurring at 5 nanosecond intervals can take many hours of computer time. The long-running simulation times negate some of the benefits of computer simulation. One approach to reducing the turnaround time for simulations is to apply parallel processing to the problem.

Research in distributed and parallel processing as applied to simulation has been progressing in recent years. Parallel processing is the use of multiple processing units to execute a program and solve a problem. These processors may use a system wide memory allowing the sharing of data, or may be connected via a communication network allowing messages to be exchanged between processors. This thesis concentrates on a distributed simulation approach, using a system composed of distributed processors connected via a communication network.

Distributed simulation is aimed at decreasing the execution time of simulations, and is ideal for application to large circuit designs. Chandy and Misra have developed several algorithms for controlling and synchronizing a simulation running on multiple processing elements (3). Also, Jefferson has developed a different approach, called the Time Warp system, for solving the same problems of synchronization and control (14). AFIT has been actively exploring distributed simulation techniques (18) in other application areas.

1.1 *Problem Statement*

The advances in the size and complexity of VLSI and VHSIC circuit designs have put an ever-increasing burden on the design tools used to document and verify the circuits before manufacture. Current circuit designs already require large amounts of computing time and power to simulate. Future circuit designs will make further demands on computing resources and simulation techniques. One approach to meeting these demands is to distribute the simulation of a circuit over the multiple processing-elements available in a distributed computer. This allows multiple processors to work together to simulate a circuit much larger than a single processor could manage. The current AFIT VHDL tool set provides a vehicle to research the applicability of distributed discrete-event simulation to circuit simulation. The use of distributed simulation provides the potential for simulating very large circuits in a time unobtainable by serial computers.

This thesis investigates approaches to distributed simulation of VHDL design descriptions using the VHDL tool set previously produced by AFIT. The goal is to develop and test a prototype distributed simulation kernel and test it with several VHDL circuits to measure the performance and efficiency of a distributed VHDL simulation implementation.

1.2 *VHDL Research at AFIT*

Past research at AFIT has led to the development of an analyzer and simulator for VHDL. Both of these tools are part of the AFIT VHDL Environment (AVE). The graphical interface front-end (19) of AVE will be usable with a parallel simulator, since the analyzer will not be changed to implement the parallel simulator. The analyzer and simulator are both written in the C programming language and hosted on UNIX-based minicomputers. The analyzer produces a compact intermediate representation of the VHDL source program. The simulator BUILD phase uses the intermediate representation and a library of simulation support routines to produce an executable program that conducts the simulation. The intermediate representation file can be used by an alternative, parallel simulator to build a parallel version of the simulation. Executing the parallel version on the Intel Hypercube will produce the same results as the existing simulator on the serial UNIX computers.

1.3 *Simulation Research at AFIT*

Distributed simulation research at AFIT has focused on the Chandy-Misra algorithm and its applications to battle management and queueing network simulation. The Chandy-Misra method ensures the synchronization of multiple processors executing independent portions of a simulation by constraining the advances in simulated time at each processor. Mannix's work measured the performance capabilities of an enhanced Chandy-Misra algorithm, called the distributed event list algorithm, on the Intel Hypercube (18). Mannix tested several variations of the Chandy-Misra algorithm and their performance on differing network topologies. Some topologies are ideally suited to the Chandy-Misra approach, while others incur too much overhead to be effective.

1.4 *Standards*

VHDL is governed by the IEEE standard 1076 (13). This document describes the syntax and semantics of every VHDL statement and also the requirements that a simulator must meet. The existing AFIT VHDL analyzer (1) already supports a subset of IEEE-1076 VHDL statements. Since the parallel simulator will operate from the intermediate format produced by the analyzer, the statement types supported by the analyzer will be available in the parallel simulation. The IEEE-1076 standard also describes the simulation cycle and the rules for propagating changes in signal values. These requirements should be met by the distributed simulation kernel.

1.5 *Approach*

The basic approach used to develop the parallel VHDL simulator consists of five steps. These steps are:

1. Identify the functionality required of a VHDL simulation according to the IEEE standard. The IEEE standard describes in detail what the actions of each statement must be. Chapter III reviews the required functions of a VHDL simulation.
2. Use the functional requirements in the standard to guide the design and implementation of the simulation control mechanism. The distributed kernel must provide

general purpose services that can be used by most VHDL statements. The design and implementation of the distributed VHDL simulation kernel is covered in Chapters IV and V.

3. Test the simulation using a variety of test cases. The range of test cases will provide a picture of the subset of VHDL supported in the distributed kernel. The coverage will, by necessity, be limited by the subset of VHDL supported in the analyzer tools. The test cases used to exercise the simulation kernel are presented in Chapter VI, while the VHDL source code is listed in Appendix A.
4. Measure the performance of the parallel simulation, both in efficiency and speed-up. Performance measurement of the test cases is also covered in Chapter VI.

1.6 Scope

This thesis covers identifying the required function of a VHDL simulator, implementing a distributed simulation kernel, and testing this distributed system with several VHDL models. The development of a new BUILD program to automatically compile the AFIT intermediate representation into executable programs for distributed simulation will not be implemented. The issues of distributed logic simulation, including performance measurement, will be studied in testing the distributed kernel.

II. Distributed Simulation Techniques

This chapter reviews available literature in the field of distributed simulation. The optimistic methods of Jefferson (14) and the conservative techniques of Chandy and Misra (6) are described. Performance studies by other researchers, including work at AFIT, are also presented.

2.1 Simulation Techniques

The simulation of physical processes has been one of the major uses of computers for many years. Experiments with applications using single processor machines have resulted in the development of more detailed and time-consuming simulation programs. These large simulation programs can take many hours to run, so it is natural to try to put multiple processors to use on a single simulation.

2.1.1 Traditional Simulation The two main categories of simulation are continuous (time-driven) and discrete-event simulation. Time-driven simulation is characterized by small, regular advances in the simulation clock. After each advance any actions scheduled for the current time are simulated; if no actions are necessary the clock simply advances to the next time interval. Event-driven simulations use a clock that jumps ahead to the future time at which the next event is scheduled. A discrete-event simulator operates by repeatedly fetching events from a queue and simulating them. As an event is simulated, new events may be created and inserted into the event queue at the appropriate time (20:39). If events are far apart or irregularly spaced, then discrete-event simulation allows the simulator program to skip over time intervals where no events occur.

2.1.2 Distributed Processing In order to overcome the speed limitations of single processor computers, parallel computers have been built. One type of parallel computer consists of multiple independent processors connected via a communications network. Each processor may have a process executing on it, and the network supports communications channels between individual pairs of processes (20:51). These channels allow processes to send messages to one another. Each message takes a finite time to arrive at the recipient.

Two messages sent from processor P_i to processor P_j are guaranteed to arrive in the order in which they were sent, but two messages sent from different processors, say P_{k_1} and P_{k_2}, are not guaranteed to arrive at P_j in the order in which they were sent. This is because an arbitrarily long time may delay one message, since there is no master processor capable of coordinating the message flow. Therefore, the only guarantee on the order of message arrivals is for the simple two process case above. This model of distributed computation is based on Hoare's communicating sequential processes (12).

2.1.2.1 Performance Measures Two important performance measures commonly used in parallel computing are *speed-up* and *efficiency.*

> The speed-up achieved by a parallel algorithm running on p processors is the ratio between the time taken by that parallel computer executing the fastest serial algorithm and the time taken by the same parallel computer executing the parallel algorithm using p processors. The *efficiency* of the parallel algorithm is the speed-up divided by p (23:43).

Some authors use a different definition of speed-up, namely: The formula for speed-up is T_1/T_p, where T_1 is the execution time using 1 processor, and T_p is the execution time using p processors (24:11). Since an algorithm tuned for single-processor execution is faster than a distributed algorithm executing on a single-processor, speed-up measured in this way is really an upper-bound on the true speed-up (24:11).

2.1.2.2 Distributed Simulation A distributed computer can support either time-driven or discrete-event simulation. Event-driven simulation has been more often used in distributed simulation, so only event-driven techniques will be considered. In distributed simulation, available techniques can be classified as *optimistic methods* or *conservative methods.* The classification comes from the method used to manage the system simulation clock. Since there is no common memory space accessible to different processors in a distributed system, each processor must maintain its own copy of the simulation clock. Much of the overhead of distributed simulation involves coordinating and updating the clock variables for each process, each of which is likely to be slightly different. Optimistic methods allow each processor to proceed at its own pace by allowing events to occur

out of order. Out of order events are corrected by rolling back the simulation when the ordering problem is detected. Conservative methods allow a processor to advance its clock only when it is absolutely sure no events will occur in the intervening time (15).

2.2 Optimistic Simulation Systems

The best example of an optimistic simulation system is the time warp system by Jefferson (14). The time warp system operates on the concept of 'virtual time.' Each process running on its own processor advances its own simulation clock (virtual time) and simulates events at its own pace. Communication between processes is used both to schedule events and for synchronization. When a message is received from an earlier simulation time (because the sending process is farther behind in simulation time than the receiver), the receiver must stop and roll back its computation, returning to the state it was in at the simulation time of the newly-received message. The receiver then simulates the effect of the new message and continues. Every intermediate message must then be simulated again. This requires each processor to keep track of past simulation events and both in-coming and out-going messages in order to be able to restore the system to an earlier state (14:412). When a process restores a prior state, it must be able to 'retrieve' any messages that it has sent, undoing their effects. The messages are 'retrieved' by sending an *anti-message* to the recipient of the original message. When an anti-message is paired with its corresponding message in the message input queue of a process, the pair is removed from the system (14:414). If the corresponding message has already been processed, the anti-message may cause the receiver to roll back. When the receiver process rolls back, it sends anti-messages to other processors, which may cause them to roll back. Time warp is an effective system when roll back operations occur relatively infrequently.

2.2.1 Drawbacks of Time Warp The biggest drawback of the time warp approach is the need to record all in-coming and out-going messages. A message or anti-message may arrive at any time during the simulation and force the processor to roll-back to the beginning of computation. Also, any messages that would cause an irrevocable action (such as writing to an output device) must not occur until the system can guarantee that

the message will never not be recalled. Since it is obviously impractical to maintain every message and anti-message for the life of the simulation, Jefferson has developed a global control method that reduces the space requirements. The global control maintains the Global Virtual Time (GVT) of the simulation (14:417). GVT is defined as:

> GVT at real time r is the minimum of (1) all virtual times in all virtual clocks at time r, and (2) of the virtual send times of all messages that have been sent but have not yet been processed at time r. (14:417)

The value of GVT never decreases, even though individual processors may roll back their local clocks. Therefore, GVT can be considered the virtual clock for the entire network of processors, and any event with virtual time less than GVT will never be rolled back. Therefore, there is no need to retain messages earlier than GVT in the log. An event that would cause output to be generated can be handled by not generating the output until GVT advances past the virtual time of the event. This ensures that the output-causing event will not be rolled back. The global control of a time warp simulation periodically computes GVT and events earlier than GVT can be removed from the log of messages. When these events are removed, any output can also be generated. The performance and space requirements for a time warp simulation depend of the frequency of computing GVT. If GVT is computed frequently to reduce space requirements, then performance will suffer due to the overhead. If GVT is not computed frequently, allowing the processors to advance their clocks freely, then a large amount of space is required to record all in-coming and out-going messages subject to roll-back.

2.2.2 Performance of the Time Warp Algorithm Chawla (7) has implemented a VHDL simulation using the time warp system for control. Chawla measured performance of the simulation in an emulated multiprocessor environment on a Sun workstation. Each module in the VHDL description is assigned to an emulated processor and three additional processors are used for control functions (7:3–4). This assignment of modules to processors avoids the problems of load balancing and limitations on the number of processors encountered on a real multiprocessor system. Chawla measured the overhead due to roll back and performance for two different VHDL models. The fraction of the total run time

absorbed by rollbacks varied between 5 and 20 percent of the total simulation time (7:4-5). This doesn't account for the time spent in simulating events that are consequently thrown out due to roll back (look-ahead computations). Chawla claims a speed-up of between $0.8N$ and $0.95N$ where N is the number of (emulated) processors. However, this is computed assuming 0 communications overhead time and, again, not accounting for the look-ahead computations. Since Chawla simulated an ideal multiprocessor system with no communications overhead, these results will need to be verified on actual hardware.

Chung and Chung (8) have taken an interesting approach to logic simulation using the time warp technique. Their approach involves a data-parallel implementation on a Connection Machine. A Connection Machine (CM) uses between 16384 to 65536 processors, but it is only a single-instruction stream, multiple-data stream (SIMD) machine. A SIMD machine must execute the same operation in all processors at the same time. This is in contrast to multiple-instruction stream, multiple-data stream (MIMD) machines like the Intel Hypercube, which may execute completely different instruction sequences in each processor node. Chung and Chung focus on behavioral simulation instead of the more common circuit or switch level simulation (8:99). To satisfy the constraints of the SIMD hardware, all circuit elements are computed via table-lookup which allows a single instruction stream to compute different results in different processors depending on the values in the table (8:99). A disadvantage of this approach is the low utilization of the event processors. An important advantage of the Connection Machine is the direct support for computing global virtual time (GVT) using the CM's *global-minimum* operations (8:103). Also, the space overhead of the time warp system can be reduced by computing the lower bound of rollback at each processor (8:102).

2.3 Conservative Simulation Systems

Development of conservative distributed simulation systems was initiated independently by Bryant (2) and Chandy and Misra (4). In the Chandy-Misra method, processes are not free to advance their simulation clocks unless they are sure that no messages from an earlier time will arrive. This eliminates the need to roll back the computation or to save large quantities of state information.

2.3.1 The Chandy-Misra Method The Chandy-Misra method uses the concept of logical processes to model a network composed of physical processes. Each logical process models the action of a single physical process from the real world. One or more logical processes are assigned to a computing node in a distributed computer (3:198). The physical interaction between physical processes is modeled by messages sent between logical processes. For the remainder of this section, the terms 'logical process' and 'process' are used interchangeably. In the Chandy-Misra method, a process' position in the network is fixed and the processes that it communicates with are also known. Chandy and Misra assume no buffering of messages takes place, so that the sender must wait for the receiver to get the message before continuing; their method can easily be extended to use buffered messages (3:199). The set of processes that send messages to process P is called the dependent set of P. Process P is said to wait for a message from the processes in its dependent set. Messages have at least two components: the simulation time of the sender, and some other data used by the simulated system (4:442).

As the simulation progresses, there is a sequence of messages sent between each pair of communicating processes. The k-th message can be represented by the tuple (t_k, m_k). There are three conditions that the sequence of messages must meet:

- The t_i values must be monitonically increasing (i.e. $0 \leq t_1 \leq t_2 \ldots$).
- The physical process corresponding to the sender must have sent message m_k to the receiver's physical process at time t_k. This means that the message in the simulated system must correspond to a message in the physical system.
- No other messages are sent between the two physical processes (3:199).

The net effect of these conditions is that if a logical process sends a message (t_k, m_k), then all messages between the physical processes up to time t_k have been simulated.

Since each logical process is independent, it must maintain its own copy of the simulation time clock. This local clock must be advanced in such a way as to maintain the conservative nature of the Chandy-Misra method. The local simulation clock T_i at logical process P_i is subject to the following rule:

All subsequent messages (t, m) sent or received by P_i must have $t > T_i$ (3:200).

This rule ensures that a message from the logical process's 'past' cannot arrive. Chandy and Misra point out the other implications of this rule as well. The logical process P_i must have received a message along each input line (from every member of its dependent set) with the t component of the message greater than or equal to T_i. Also, logical process P_i must ensure that it will not send a message (t, m) to another logical process where t is less than T_i (3:200).

In order to keep track of message times, each process must compute a clock-value for each of its communications channels. The sending and receiving processes along a given channel will independently compute the clock-value of their communications link. However, the two values will normally be different, since some events may have been sent and not yet received. The receiver's channel clock-value is the t-component of the last message received across the line (3:200). The process can then set its local clock (T_i) to the minimum of all the channel clock values (20:52). In some cases, the process may be able to advance its clock past this minimum if it can deduce the actions (messages sent) of the corresponding physical process (20:52).

Since the advancement of a logical process's simulation clock is constrained as shown above, the process must alternate between computing and waiting to communicate. The computing phase of a process corresponds to simulating events and generating new events, while the time spent waiting to communicate corresponds to synchronization overhead. Chandy and Misra give the waiting rules for logical processes: a process must wait for an incoming message on all input lines whose clock values are less than or equal to the logical process' clock (T_i). When a message is received or sent, the logical process enters its computing phase and determines which events can be simulated or messages can be sent. The computing phase will end when the process begins waiting on new set of message lines.

2.3.2 Deadlock Deadlock is possible in a distributed system. In the Chandy-Misra distributed simulation technique, deadlock arises in two different ways. The first method is demonstrated in Figure 2.1. If the physical processes at point B routes every message from the source to Proc1 and no messages to Proc2, then the node at M will never receive any

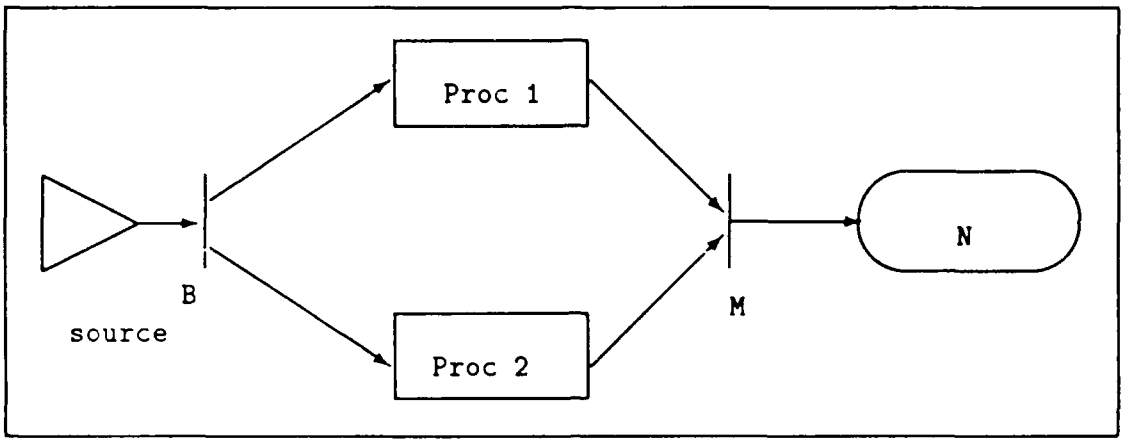

Figure 2.1. A distributed simulation that does not progress (20:56)

messages from Proc2. Node M's channel clock-value for the Proc2 link will remain equal to 0, while the channel clock-value for the Proc1 link will continue advancing. By the rules above, the simulation clock (T_s) for node M will equal 0 forever; node M cannot send any messages to the sink at N, and the simulation cannot advance (20:55). The other problem that produces deadlock is cyclic waiting. Cyclic waiting occurs when a cycle of processes are waiting for a message that will never come. In Figure 2.2, suppose that none of the processes x, y, and z will send a message without receiving one first. The numbers on each arc are the values of the channel clocks. Figure 2.2 actually demonstrates one special condition that can cause cyclic waiting. Notice that process y has taken in a message, advancing its channel time to 20, and hasn't generated a corresponding output message. Since z has processed this message without sending an output message, the cyclic waiting starts. Process x is waiting on z, z is waiting on y, and y is waiting on x, therefore, no messages will be sent and the system deadlocks (20:55).

The problem of deadlock is one of the major concerns in distributed processing. There are two approaches to solving this problem: *deadlock avoidance*, and *deadlock detection and recovery*. Several researchers have been involved with measuring the performance of various types of deadlock resolution (30, 24, 11).

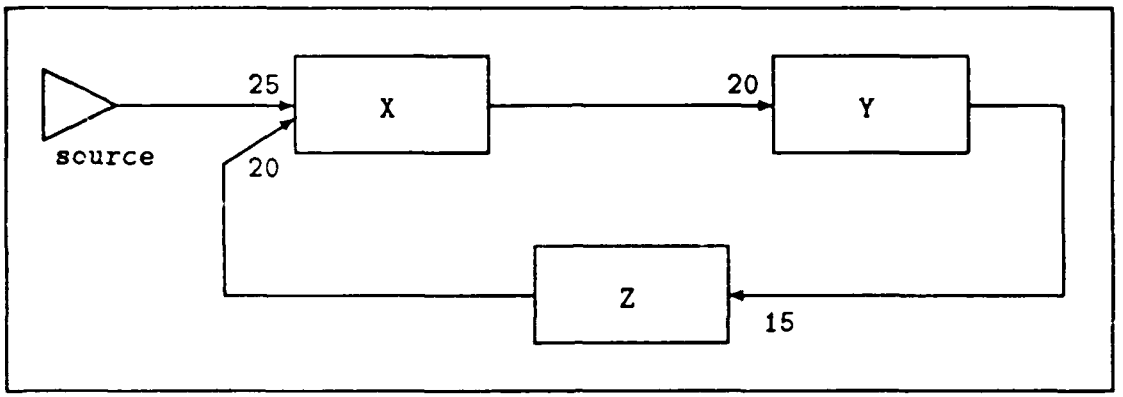

Figure 2.2. A distributed simulation that deadlocks (20:56)

2.3.2.1 Deadlock Avoidance In their initial paper, Chandy and Misra proposed using *null messages* to eliminate deadlock in the simulation code (4:446). A null message contains just the sender's simulation time, and no other data $(t, NULL)$. A null message is sent between two processes to denote that the sender process will not send any other messages to the receiving process before the time contained in the null message (3:201). Therefore, a null message serves to update the receiver's clock time.

Null messages can very easily eliminate deadlock of both types described above. Whenever a router sends a message to only one of its possible outputs, it sends a null message to the other outputs to update the channel clock-value of the recipient. In the example in Figure 2.1, whenever B sends a message to Proc1, it also sends a null message to Proc2 allowing it to update its channel clock. This change to a channel clock may allow Proc2 to proceed in its simulation, thus avoiding deadlock. In the cyclic waiting case (Figure 2.2), deadlock may be avoided by having process y send a null message at time 20 to update the channel clocks. Null messages eliminate the possibility that the simulation management program will deadlock. However, if the physical system is susceptible to deadlock, null messages will not allow the simulation to advance. If the physical system under simulation deadlocks, then null messages will be sent forever, and each process will advance its clock (T_i) without any events occurring (20:58).

However, in some cases the number of null messages necessary to prevent deadlock is

very large. Mannix has observed a thousand-fold increase in the number of null messages between slightly different model configurations (18:4–20). This suggests that, for some simulation models, the Chandy-Misra approach will have too much overhead (in the form of null messages) to be efficient.

2.3.2.2 Deadlock Detection and Recovery Chandy and Misra and others have recognized the possible problems with the null message method of avoiding deadlock. In a later paper, Chandy and Misra propose running the simulation until it deadlocks, and after detecting the deadlock, a master process resumes the simulation (3). This system requires a distributed means of detecting deadlock, since a single processor devoted to detecting deadlock would become the bottleneck of the simulation. Chandy and Misra have solved the problem of detecting deadlock using a distributed method in (5). Several variations of deadlock detection and recovery techniques have been studied recently (24, 30). Chandy and Misra's method uses three criteria to determine if a set S of processes has deadlocked. These are:

- All processes in S are idle.
- The dependent set of every process in S is a subset of S; and
- There are no messages in transit between processes in S (5:148).

Chandy and Misra's algorithm determines if a portion of the system has deadlocked, and the master process issues the messages needed to restart the simulation. The drawback to this method is the requirement for a master process that may become the bottleneck in the simulation. Also, the deadlock conditions will occur much more frequently using this method of simulation control, so a large portion of the processing time may be spent recovering from the deadlock (15:16).

2.3.3 Performance of The Chandy-Misra Algorithm One of the early studies using the Chandy-Misra algorithm was by Reed and Malony (24). They conducted experiments using a shared-memory implementation of the Chandy-Misra algorithm. Shared memory allows message-passing to be implemented by accessing a region of memory reserved for the

message queues (24:9). Mutual exclusion between processes accessing this region is required to prevent the storage area from becoming corrupted. Reed and Malony experimented with both deadlock avoidance and deadlock recovery variations of the basic algorithm. The null message algorithm is used for deadlock avoidance (3). Their deadlock recovery technique takes advantage of the shared-memory implementation to use a "guardian processor" to detect deadlock (24:10). The guardian processor is able to monitor the state of all the logical processes, and when deadlock is detected, recovery is made and the simulation resumed. Reed and Malony measured the speed-up of their distributed simulation in relationship to the distributed algorithm executing on a single-processor "distributed" computer. This definition of speed-up, as used by Reed and Malony, is really an upper-bound on the true speed-up (24:11).

Reed and Malony simulated queuing networks and measured the performance of both variations of the Chandy-Misra algorithm on two different networks. The central server network is a small model that can be analyzed easily, and the performance of the deadlock recovery algorithm is poor. The speed-up is near one (implying virtually no speed-up), since the network deadlocks after each message and is forced to recover (24:11). The simulation using deadlock avoidance shows good speed-up; however, this result depends on the ratio of null messages to event messages.

Reed and Malony make one important observation, "a single processor implementation of the Chandy-Misra algorithm is almost always slower than an equivalent, event-driven implementation" (24:11). This is due to the overhead of null message handling.

In previous work at AFIT, Mannix (18) has studied the Chandy-Misra algorithm using a distributed event list approach to parallel simulation. The distributed event list maintains a queue of events on each processor that is processed in time order (18:3–14). Synchronization between processors is handled by the Chandy-Misra null message technique.

Mannix also investigated two variations of the basic null message deadlock avoidance technique described by Chandy and Misra. The first variation on Chandy-Misra is the transmission of additional null messages (called stimulus nulls) strictly to improve

throughput. These messages are transmitted after the execution of a certain number of events, specified as a ratio of events to stimulus nulls (18:3–36). As the stimulus null ratio is increased beyond a certain point, the incremental increases in speed-up diminish rapidly as the system becomes swamped in null messages (18:3–37). The other variation in null message handling is called the time-out algorithm. This method was suggested by Misra (20) and involves delaying the transmission of a null message until a specified amount of processing time has elapsed (18:3–37). This delay is hopefully made up for by sending an actual event message instead of the null message.

Mannix ran simulations of queuing networks of three different topologies. These simulations ran using from 1 to 32 processors on an Intel iPSC/1 computer. The *tandem* topology arranges the logical processes into a chain, with each process receiving events from a single source and sending events to a single destination. This is a simple topology that doesn't stress the distributed simulation control technique (18:4–14). This simple topology shows excellent performance in distributed simulation; Mannix observed linear speed-up, using the distributed event list algorithm, for any number of processors. The *feed-forward* topology doesn't have any feedback loops, but each logical process may send events to more than one recipient or wait for events from more than one sender. This topology also showed linear speed-up under the distributed event list method (18:4–16). The tandem topology was modified slightly for the third configuration. This added a *pseudo-feedback loop* from the end of the tandem chain to the source logical process. While no events were ever sent along this link, it drastically increases the number of null-messages processed by the distributed network. This additional change caused a hundred-fold increase in execution time, and an increase in the number of null messages from 6422 to 1.9×10^6 (18:4–20).

The feedback networks had too many null messages generated by either null message variant to complete the simulation tests. (18:4–23). A small increase in speed-up is possible using stimulus nulls for both the feed-forward and tandem networks (18:4–27). However, increasing the percentage of stimulus nulls in the feed-forward network beyond 10% caused the system to saturate and decreased speed-up over the distributed event list baseline (18:4–27).

The time-out algorithm variant shows a decrease in performance for both the tandem

and feed-forward networks (18:4-24). This is partly related to the high internal event—communications ratio of the test model. A system that is more communications bound is expected to show more benefit from the time-out algorithm.

An analysis of the Chandy-Misra algorithm as applied to logic simulation has been done by Soule and Gupta (28). They tested a deadlock detection and recovery version of Chandy-Misra using several large scale circuit designs. Their primary measurement is the *concurrency* of the simulation. This is the speed-up obtainable assuming arbitrarily many processors, no synchronization costs, and no scheduling overhead (28:83). This is also a measure of the parallelism inherent in the simulation. The concurrency depends on the simulation control algorithm (in this case, Chandy-Misra) and the circuit model under test. One of Soule and Gupta's test cases is the vector control unit from the Ardent Titan supercomputer. This circuit represents a large synchronous gate array (28:82). Three other circuits of different characteristics were also modeled. The synchronous nature of the control unit is reflected in the plots of parallelism versus simulation iteration (the execute, deadlock, recovery cycle). Immediately on the edge of the system clock, a large number of concurrent events occur and are simulated. As these signals propagate through the combinational logic of the simulation, relatively few events need to be simulated. Finally, the cycle repeats itself with the next clock activation (28:83). The pipelined architecture of the Ardent vector unit results in very steep peaks in parallelism with very little activity between clock pulses. This is due to the very small amount of combinational logic between registers in the Ardent pipeline. Another test case, a 16-bit multiplier, shows greater activity between clock pulses because of the greater amount of combinational logic. The average concurrency of the Ardent unit is 107, while the multiplier shows a concurrency of 45 (28:83). This implies that a 107 processor system with perfect inter-processor communications (zero delay) could be efficiently used for simulating the Ardent with an execution time 107 times less than a single processor. Of course, real-world communications delays and overhead will prevent this theoretical speedup from being achieved. Earlier work by Soule and Blank (27) showed a concurrency of only 30 for the multiplier (28:83). The 1.5 factor increase in concurrency is offset by the increased complexity of the Chandy-Misra implementation. Another factor in the Chandy-Misra implementation is deadlock reso-

lution. The simulation of the Ardent spends 58% of its time resolving deadlocks. The time spent resolving a single deadlock is the same as processing 700 logic element activations (28:84). Even though the deadlock resolution takes place in parallel, the resolution limits the speed-up achieved even though the concurrency of a given model may be high.

Soule and Gupta analyze the sources of the deadlocks occurring in their simulations. The major sources are: register loads on clock pulses in synchronous circuits, multiple paths with different delays, and unevaluated paths (28:84–85). Each of these causes can be worked around through introduction of null messages into the basic deadlock detection and recovery system, or by taking advantage of logic circuit behavior to augment the Chandy-Misra techniques (28:86).

As an example of an unevaluated path and the workaround used by Soule and Gupta, consider the simulation of a simple Boolean OR gate. If one input to the gate is a 1 and the other input is a 0, then the output must also be a 1. If an event occurs on the other input, changing its value to 1, then no event occurs on the output of the OR gate, since its value remains 1. Deadlock can occur in later stages of the circuit, since the channel clock times along this path are not updated. As one solution to this general problem, Soule and Gupta introduce null messages into their basic deadlock detection and recovery scheme. They offer no advice on the frequency of sending these extra nulls, other than to say "we need to be very selective about which elements should send null messages" (28:86). Of course, the simple case of an OR-gate can be worked around by taking advantage of the circuit and propagating the input change to the output signal.

Su and Seitz (30) have done evaluation of several variations of the Chandy-Misra algorithm in logic circuit simulation. Their design goal is based on the following convoluted observation:

> The principle trouble with naive implementations of conservative Chandy-Misra distributed simulation programs in any situation in which processing null messages is as costly as processing event-containing messages is that the volume of null messages may greatly exceed the number of event-containing messages. (30:38)

In a logic simulation, processing an event takes very little computation, so null messages compose a significant fraction of the computational workload. Su and Seitz have developed several strategies for reducing the number of null messages that must be processed. Indefinite lazy message sending (30:39) is a method to defer sending null messages until an event-containing message must also be sent along the same channel. This reduces the message traffic, but deadlock must be prevented from occurring. This requires a modification to the action of a logical process. When a logical process has no input messages to process (simulate), it must take action to break a possible deadlock (30:39). This may require sending a message along an output channel, or sending a request message to a predecessor process requesting an input message.

Su and Seitz have developed six variations of the basic Chandy-Misra algorithm combining various strategies for handling null messages and event-containing messages. The resulting variations range from the standard Chandy-Misra technique to indefinite lazy sending of null messages with event messages sent in batches. Also, a demand driven variation is possible, where messages are sent only when the recipient requests them. Su and Seitz measured performance statistics for all six variations running on a multiprocessor emulation system, and on a real multiprocessor computer (the Intel iPSC/1 and iPSC/2 Hypercube systems). The performance of the six variations is roughly the same on the emulated multiprocessor, since messages have very low cost. On the Hypercubes, the standard Chandy-Misra method shows much poorer performance than the indefinite lazy message techniques (30:41). This is due to the large numbers of null messages that must be processed in the standard method. Even so, the speed up for Su and Seitz is very good. Most test cases show a linear relationship when the logarithm of the execution time ($\log_2 t$) is plotted against the logarithm of the number of processors ($\log_2 N$). In one case, a multiplier simulation on the iPSC/2 takes $2^{9.1}$ (approximately 550) seconds on 1 node, and decreases steadily to $2^{5.5}$ (approximately 45) seconds on 16 nodes (30:41). This gives a speed-up of greater than 12 using 16 processors. This same speed-up relationship also occurs on a iPSC/1 using 4 to 128 nodes.

The basic conclusions of Su and Seitz are that the Chandy-Misra method works well using the lazy message sending techniques, and that performance is limited by the

concurrency of the system under simulation (30:43).

2.4 The Dimensions of a Simulation Application

There is a very wide range of activities that can be simulated using a computer. The simulation of logic circuits is very different from the simulation of a telephone network or missile launch system. These differences are as important as whether an optimistic or conservative simulation control mechanism is used. Reynolds has developed some characteristics that can be used to classify simulation applications according to their functionality and complexity (25).

The first characteristic is *determinism*. An application may show varying degrees of determinism ranging from deterministic to nondeterministic. A deterministic simulation will give the same results when executed several times on a sequential processor. *Queuing* indicates a paradigm where the time for processing an event at a logical process is dependent on the presence of other events. Queuing occurs in network simulations where messages are queued for processing; logic simulations are one type of simulation where this doesn't occur. *Processing delays* occur if an event emitted by an logical process has a simulation time greater than the arriving event. *Causality* indicates the degree to which every event arriving at a logical process leads to an identifiable subsequent event leaving the logical process. Causality is one of the more critical characteristics that a simulation control mechanism has to contend with. *Production* occurs when an event can be created by a logical process, while *consumption* occurs when a logical process may take in an event without generating an output event. Another characteristic is how objects *change state* in a simulation: how many objects change and with what frequency. *Balance* is a measure of the uniformity of processing requirements across the logical processes. Balance has implications for ensuring that all processors are kept busy in a distributed simulation. The *level of activity* in a simulation is based on the number of logical processes that are busy at any instant of time. A low level of activity may signal little oppurtunity for parallelism in a simulation. *Connectivity* measures how much events in one logical process can directly affect events in another. This measure includes the direction of information flow in a simulation, and the topology of the inter-process communications network.

With further work, these characteristics can possibly be used to tune the simulation control mechanism for the most efficient operation. In addition, the characteristics can be used to classify a new application in order to determine what control mechanism would be most suitable for it. Chapter IV analyzes VHDL simulation in light of these characteristics.

2.5 Summary

The two primary techniques of synchronizing a distributed simulation are the optimistic methods, such as time warp, and the conservative methods, such as Chandy-Misra. Many studies have been done with varying degrees of success, including digital logic simulation using both time warp and Chandy-Misra null messages (deadlock avoidance). The results of Soule and Gupta (28) show potentially large speed-ups available on logic simulations using Chandy-Misra synchronization techniques. The characteristics of a simulation can affect the performance of a simulation control mechanism; this, in turn, affects the choice of control mechanism for a given problem.

III. The VHSIC Hardware Description Language

The VHSIC hardware description language (VHDL) has been developed to support the design and testing of large scale circuitry. Modern very-large scale integrated circuits are very complex and hard to design. In the past, each hardware vendor has been developing their own computer-aided-design (CAD) tools to improve their productivity. The result has been a mess of incompatible languages and tools (10:13). The government has pressed for the development of a standard high-level language for specifying the operation and structure of circuitry to improve the documentation of military systems. Using a standard language will also help improve productivity and allow standardized CAD tools to be built. Development of a standard hardware description language started in 1983 when Intermetrics, Inc. was awarded a contract to design a hardware description language. The specification of VHDL was completed in 1987 with the release of IEEE standard 1076, the IEEE Standard VHDL Language Reference Manual (13).

The next section is presents the tools for using VHDL developed at AFIT. This chapter includes a general overview of VHDL features and capabilities. The final section presents the IEEE standard requirements for simulation of a VHDL circuit description. VHDL is a very large language that provides many ways of stating the same functions. VHDL features range from traditional programming constructs to statements and functions that directly model hardware circuitry. This chapter emphasizes features that are unique to VHDL or requirements that impact simulating a VHDL model.

3.1 The AFIT VHDL Environment

Since 1986, AFIT has been developing an environment, consisting of a set of tools for Unix based computers, for using VHDL. The tool set includes a graphical front end, an analyzer, and a simulator (1, 9, 19, 21). The analyzer and simulator run on Unix-based computers, while the graphical front end uses a microcomputer. The AFIT VHDL environment is aimed at universities, in order to support the introduction of VHDL into a classroom environment.

The analyzer supports a large subset of the VHDL language. The analyzer performs syntactic and semantic analysis of the VHDL source code and converts it to an intermediate form, called VHDL Intermediate Access (VIA). VIA has all of the information from the source program encoded within it, and the simulator uses the VIA representation instead of the original VHDL source code.

The VIA file format consists of four components: the header, the symbol table, the operation table, and the string table (9). The symbol table contains classification and type information about each symbolic name used in the VHDL source. The operation table contains a abstract syntax tree of all of the statements of the VHDL code. This structure has all of the information necessary to be able to simulate the execution of the VHDL model. Lastly, the string table contains all of the string literals from the VHDL source program. The strings are stored here for easy reference and compactness.

The VHDL simulator reads the description of the VHDL model from the VIA file. This description is used to build a set of functions simulating the operation of the VHDL model. The set of functions are linked with a simulation support library to produce an executable program simulating the VHDL model. Therefore, running a VHDL simulation is a two-step process. First, the VIA representation of the model must be "compiled" into an executable simulator, and then, the executable simulator must be executed to produce the simulation results. The simulation program can be used either interactively, or in batch mode. When used interactively the operator can set break-points to interrupt the simulation, and view or set signal values (9). The simulator also accepts an "input stimulus file" that gives starting values for signals of interest.

The graphical VHDL user interface (GVUI) part of the AFIT VHDL environment allows a designer to draw a circuit using a personal computer. The GVUI then generates the correct VHDL code for the circuit, allowing the designer to simulate the circuit. The GVUI is expected to decrease the time it takes for a designer to become proficient with VHDL tools. Also, the GVUI automatically generates the VHDL code, reducing the possibility of errors in translating a circuit to VHDL. The GVUI supports hierarchical decomposition of the circuit design, giving the designer the capability to build a circuit from existing parts. The GVUI runs on standard IBM personal computers making it easily available for

student use (9).

The VHDL process is summarized in Figure 3.1. The designer writes the VHDL source code for a component. The source is analyzed and converted to VIA format. Next, the build phase converts the VIA format of the component into a series of C routines for simulation; these routines are combined with an existing component library, if any, and the simulation kernel. The end product is an executable simulation program.

3.2 VHDL Language Features

VHDL is a very large language. The language constructs of VHDL are chosen to allow modeling the behavior or structure of almost any circuit. This expressive power allows the circuit designer to develop prototype designs quickly, while the simulation capabilities allow the designer to test the design without building a physical circuit. Behavioral modeling allows a designer to specify the functionality of a circuit in high-level algorithmic terms. The behavioral model has no relationship to an implementation of a circuit; it is simply a functional description. This behavioral description can be simulated by itself, or in conjunction with other components in a more complex circuit. Once the functionality of a circuit has been decided upon, the designer can convert the description of the circuit to structural terms. In a structural model, the VHDL constructs describe exactly the hardware implementation of the circuit. The structural model can then be simulated, which allows the circuit to be simulated using the same design from which it will eventually be built.

The major building block of VHDL is the *design entity*. The VHDL description of a component or complete system is composed of design entities. Each entity is composed of an interface and body (16:28). The interface describes the component's external connections to other components and any other characteristics that need to be visible. The interface contains the declaration of *ports* which provide channels for communication with the entity's environment (13:1-3). The entity body describes the organization and operation of the component. An entity may have several bodies, each focusing on different aspects of the internal characteristics of the component.

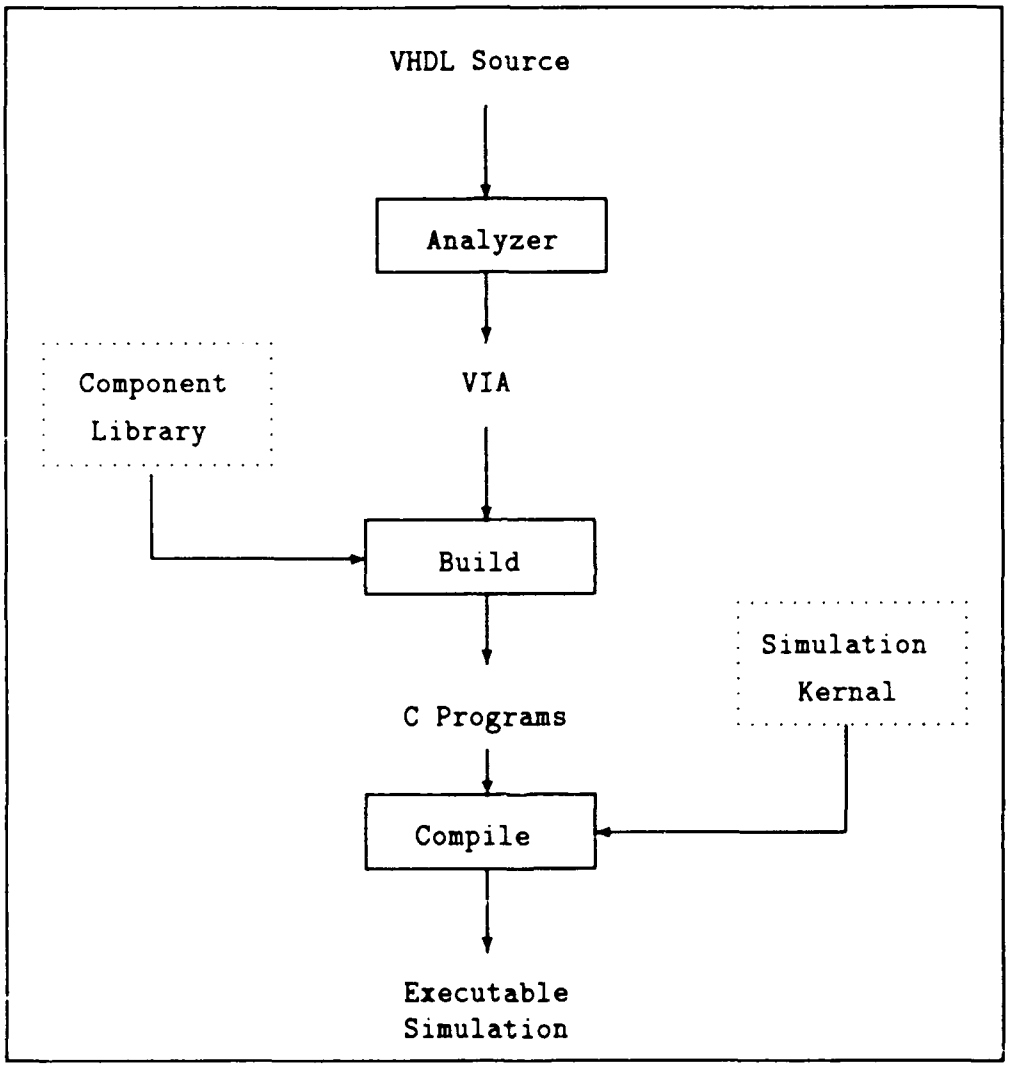

Figure 3.1. The VHDL Design Process

There are three general techniques for describing an entity. A *structural* description of an entity defines each subcomponent and its interconnections with other subcomponents and the external ports. A *data flow* description describes the flow of data through the entity and the transformations performed by subcomponents. Finally, *behavioral* description allows the designer to describe the circuit's function in a high-level algorithmic approach (16:34).

3.2.1 Packages VHDL uses the concept of a package to group related declarations and components. A package consists of two separate parts, a declaration and a body. The package declaration defines the visible contents of a package; a package body provides hidden details (13:2–1). This allows the bodies of components or subprograms to be hidden within a package body. VHDL supports the use of subprograms (procedures and functions) for converting between different user-defined types, or defining the behavior of components.

3.2.2 Declarations Declarations in VHDL appear in the declarative part of an entity. VHDL allows the user to define types, subtypes, constants, and signals as required to model a circuit (13:1–4). VHDL supports a wide variety of types: *scalar* types such as integers, floating point, physical or enumerated types; *composite* types such as records and arrays; *access* types are similar to pointers in other languages; and *file* types allow access to data sets (13:3–1). The standard language has the following predefined types: `BOOLEAN`, `BIT`, `CHARACTER`, `INTEGER`, `REAL`, `TIME`, `STRING`, and `BIT_VECTOR` (13:14–1). All of these types are scalar types, except for `STRING` and `BIT_VECTOR`, which are array types.

3.2.3 Objects Objects in VHDL are the 'variables' of the language. Objects arise from subprogram parameter declarations, entity port or generic declarations, and object declarations (13:4–3). Every object has an associated type, which may be either user-defined or one of the standard types.

There are three classes of objects: constants, signals, and variables (13:4–4). Constants are defined during elaboration (see Section 3.3.2) and cannot change in value during execution (13:4–4). Variables are identified by the keyword `variable` in the declaration.

Table 3.1. Predefined VHDL Attributes

STABLE	DELAYED	QUIET
TRANSACTION	EVENT	ACTIVE
HIGH	LOW	POS
PRED	SUCC	LEFT
RIGHT	RANGE	LENGTH

The value of a variable is changed using a *variable assignment* statement, which takes effect immediately (in simulation time).

Signals are identified by the keyword **signal** in the declaration. In addition, objects declared in port lists of an entity are also signals. The value of a signal is indirectly modified by the signal assignment statement, which affects the future values of the signal (13:4–6). Each signal has one or more associated drivers, which maintain a list of future values that the signal will have and the simulation time at which the new value will take effect. Each of these value-time pairs is called a *transaction*, and the list of transactions for a signal form its *projected output waveform* (13:8–4). As simulation time advances, the transactions in the projected output waveform of a given signal will, in succession, become the value of the driver (13:12–9).

VHDL allows the definition of attributes that denote characteristics of entities and objects. The standard also predefines some attributes that must be supplied by an implementation (13:14–1). Many attributes return characteristics of signals, or are themselves signals. For example, the attribute `S'STABLE(5 ns)` returns a Boolean value indicating whether or not an event has occurred on signal `S` within the last 5 ns. The attribute `S'DELAYED(10 ns)` returns a signal which has the same value as `S` delayed by 10 ns. The attribute `T'HIGH`, when applied to a scalar type, returns the upper bound of the type `T`. Likewise, `LOW` returns the lower bound of a type. Table 3.1 lists many of the predefined attributes.

3.2.4 Operators VHDL supports a large set of pre-defined *operators* that act on objects of various types. Operators are used to build up expressions that compute new values for signals and variables. Expressions are used in the assignment statements described in

Table 3.2. Predefined VHDL Operators

Logical	and	or	nand	nor	nor	xor
Relational	=	/=	<	<=	>	>=
Adding	+	-	&			
Sign	+	-				
Multiplying	*	/	mod	rem		
Miscellaneous	**	abs	not			

Section 3.2.5. The user may define new operators to act on user-defined types. The predefined operators are divided into six classes: logical, relational, adding, sign, multiplying, and miscellaneous operators (13:7–2). The actual operators are listed in increasing order of precedence in Table 3.2. The logical operators are defined for the type **BIT** and **BOOLEAN** as well as one dimensional arrays with **BIT** or **BOOLEAN** elements (13:7–2). The relational operators test for equality or inequality. Testing for equality is defined for any type; while testing for inequality (less than, greater than, etc.) is defined only for scalar or discrete array types (13:7–3). The adding and multiplying operators are defined for any numeric type, with the exception that **mod** and **rem** are defined only on integers (13:7–6). The **mod** operator implements the modulus function, while the **rem** operator computes the remainder of its operands. The miscellaneous operators implement exponentiation, absolute value, and Boolean negation. Absolute value and exponentiation operate on any numeric type, but only integral exponents are allowed (13:7–8).

3.2.5 Statements There are two classes of statements within VDHL: sequential statements and concurrent statements. *Sequential statements* make up the bodies of VHDL subprograms (13:2–4). *Concurrent statements* are used to define interconnected blocks and processes that describe the overall behavior or structure of a design (13:9–1).

3.2.5.1 Sequential Statements There are several types of sequential statements: the wait statement, assertion statement, signal assignment statement, variable assignment statement, procedure calls statement, if statement, case statement, loop statement, next statement, exit statement, return statement, and null statement (13:8-1). The if, case, loop, next, exit, return and null statements are sequential programming constructs

```
X <= A and B after 13 ns;
```

Figure 3.2. A Simple Signal Assignment Statement

common to many programming languages. The variable assignment statement is used to update variables in sequential blocks of code. A variable in VHDL acts like variables in other computer languages, and the changes made by assignment statements take effect immediately. The other statement types have special features in VHDL. The wait statement allows the VHDL program to suspend itself until a signal changes value or a certain simulation time is reached. The assertion statement allows the program to catch and report errors based on the values of signal or variables during simulation.

The most important statement is the signal assignment statement. This statement can either be used sequentially or concurrently. It is used to modify the projected output waveform of a signal (13:8-3). The signal assignment statement takes one of two forms, depending on the type of delay semantics desired. Transport delay is specified by the keyword `transport` in the statement, while the default is inertial delay. The differences between the two forms of delay is covered in Section 3.3.1. An simple signal assignment statement appears in Figure 3.2. This statement constructs a new transaction in the projected waveform for signal `X` at 13ns in the future, with a new value corresponding to the Boolean-AND of signals `A` and `B`. This example defaults to inertial semantics, since the `transport` keyword was not specified. The change made by this signal assignment does not take effect until the specified delay time (13 ns in the example) has elapsed. The signal assignment statement causes a transaction to be entered into the projected output waveform, for signal `X`, with a new value at 13 ns into the future.

3.2.5.2 Concurrent Statements One of the most useful features of VHDL is the capability of modeling concurrent aspects of hardware. Real hardware components act in parallel with each other and not in a strictly sequential manner. Concurrent statements are used for component instantiation, modeling concurrent processes, and generating repetitive blocks of VHDL code (13:9-1). The block statement is used to group together

concurrent statements to support hierarchical decomposition of an entity (13:9–1). The generate statement is used to define iterative or conditional elaboration of a portion of a description (13:9–13). This is similar to macro processing in other languages, and allows related statements to be generated automatically.

A *concurrent process statement* defines a sequential process that computes some function and assigns the results to signals within the model (13:9–3). A process has a *sensitivity list* of signals which cause the outputs of the process to be recomputed. Whenever an event occurs on one of the signals in the sensitivity list, the process must be executed and new output results computed.

The signal assignment statement can also be used as a concurrent statement. A concurrent signal assignment statement is equivalent to a process statement which assigns values to the same signal (13:9–7). A signal assignment may also occur in two variants, the *conditional assignment* (13:9–8), and the *selected assignment* (13:9–9). The conditional signal assignment corresponds to an if-then-elsif-else statement within a process. The selected signal assignment corresponds to a case statement within a process.

3.2.5.3 Sensitivity A concurrent signal assignment statement is sensitive to the signals listed in the expression on the right hand side of the statement. This corresponds to the sensitivity list of the process statement. The sensitivity list determines when the process must be activated to compute new values.

Figure 3.3 shows a concurrent signal assignment statement and an equivalent process statement. Since a process statement contains only sequential code, the signal assignments on the right are sequential signal assignment statements. Both the process and the concurrent signal assignment are sensitive to signals A and B. Whenever A or B change values, the value of S must be updated. The change in value of A or B results in the activation of a process to recompute S. The process may either derive from the explicit process statement or implicitly from the concurrent signal assignment statement.

3.2.6 Component Instantiation Since most real-world hardware is composed of components and sub-components that are wired together, VHDL supports this idea in

```
S <= "01" when A < B else        process(A,B)
     "10" when A > B else        begin
     "00";                          if A < B then
                                       S <= "01";
                                    elsif A > B then
                                       S <= "10";
                                    else
                                       S <= "00";
                                    end if;
                                 end process;
```

Figure 3.3. Signal Assignment Statement Versus Process Statement (21:3–10)

```
entity full_adder is
     port ( x, y, cin :  in BIT; sum, cout :  out BIT );
end full_adder;
```

Figure 3.4. A Full Adder Entity

modeling hardware. This allows a designer to model a system using a library of pre-tested components. A component is a VHDL entity with a defined interface composed of ports. The ports indicate the connections that must be made for this component to be used as part of a larger system. For example, an entity (or component) representing a full adder with three BIT inputs and two BIT outputs is shown in Figure 3.4. The port list defines three ports of mode **in** and two ports of mode **out**.

A *port* is a special type of signal, which defines the interface between components. A port has both a mode and a type. The mode of a port constrains the direction of information flow. A mode of **in** indicates that information flows into the component, while **out** indicates the data flow goes out from the component. A port may pass information in both directions using a mode of **in out**. The type of a port corresponds to one of the standard predefined types or to a user-defined type. The type specifies the set of values a port may assume (16:30).

Once the entity definition of a component is entered into the design library, it may be instantiated as part of other components. This is done with the *component instantiation*

```
architecture structural of test is
component adder
   port(x, y, cin :  in BIT; sum, cout :  out BIT);
end component;

for all :  adder use entity full_adder(anarchitecture)
              port map( x, y, cin, sum, cout);

signal a, b, c :  BIT;
signal s, c2 :  BIT;

begin
   an_adder :  adder port map( a, b, c, s, c2 );
end structural;
```

Figure 3.5. Instantiating the Full Adder Component

statement. This is another concurrent statement, which defines the signals connected to each port of the component (13:9–10). A component may be instantiated multiple times within a given entity with different signal connections for each instance of the component. To use the component in Figure 3.4, the statements in Figure 3.5 could be used. Figure 3.5 defines the body (or architecture in VHDL terms) of an entity named **test**, which declares five signals and instantiates one component. The component statement defines a name (**adder**) that is used as a 'type' in the declaration of the actual component (**an_adder**). The component configuration statement (the statement beginning with **for**), defines the entity from the design library which will be used to simulate all the instances of component 'type' **adder**. The port map clause in the definition of component **an_adder** lists the signals that are connected to each port in the component. The same component type (**adder**) could also be instantiated with different connected signals, if more signals were declared within this entity.

Component instantiation can be used by a designer to build up complex structures in VHDL, just like hardware systems are built from standardized components.

3.2.7 Summary The language constructs of VHDL are chosen to allow modeling the behavior or structure of almost any circuit. This expressive power allows the circuit

designer to develop prototype designs quickly, while the simulation capabilities allow the designer to test the design without constructing it.

3.3 Simulation of VHDL

One of the benefits of modeling hardware with a language like VHDL is the ability to simulate the hardware design without building it. Simulation can be used to uncover design flaws and to experiment with new versions of hardware very inexpensively.

One of the difficulties in simulating VHDL is the wide range of styles that may be used in writing VHDL code. A structural description of a component will have a large number of sub-components and gate-level descriptions. In simulation, this results in a very large number of very simple processes. Conversely, a pure behavioral description may be a few very large process statements that execute large blocks of sequential code. A data flow description is composed of concurrent signal assignments, which results in a relatively large number of processes of moderate complexity. The design tradeoffs of a simulator must take into consideration these conflicting design styles.

The IEEE standard for VHDL also defines the requirements for successfully simulating a circuit described in VHDL. There are two facets to simulation, the *elaboration* of a circuit model, and the actual *execution* of the model. Elaboration of a model ensures that the declarations within the model achieve their effects (13:12–1). The model is simulated by repeatedly executing the simulation cycle.

3.3.1 Timing Semantics in VHDL Some early hardware description languages used *anticipatory* semantics for signal assignment statements (17:10). For example, the ADLIB statement **assign** X **to** Y **delay** t; has the effect of assigning the value X to Y at t units in the future and cannot be overridden before time t. This type of semantics is troublesome when a component has a different propagation delay (value of t) when transitioning from low to high, than when transitioning from high to low. In this case, a quick pulse would be propagated to the output in simulation, when the actual device is unaffected by the pulse (17:11). Thus, the anticipatory semantics incorrectly model the circuit.

VHDL uses another model of signal assignment, called *preemptive semantics.* In this model, predictions of future events (changes in values of signals) can be overruled or preempted by other predictions. VHDL utilizes two different variations of preemptive semantics called *transport delay* and *inertial delay.* The default method is inertial delay, which is characteristic of switching circuits: a pulse whose duration is shorter than the switching time of the circuit is not transmitted to the output (13:8–4). Transport delay is characteristic of devices such as transmission lines where any pulse is transmitted no matter how short its duration (13:8–4). The proper use of transport and inertial delays ensures that the VHDL model accurately follows the physical device.

VHDL also uses a infinitesimal delay unit, called *delta delay.* Whenever 0 delay time is selected for a given statement, VHDL uses delta delay as the actual delay time. Delta delay is a recognition that hardware takes some time to change state (16:34). Delta delay is greater than zero but smaller than the smallest measurable unit of time (21:3–17). The effect of delta delay on a simulation is to order VHDL events that take place at the same simulation time. Figure 3.6 illustrates the use of delta delay in simulating the two statements at the top of the figure. The two VHDL statements will use 0 ns for the delay time since no explicit time is given. Therefore, VHDL uses delta-delay to separate the events. The rising edge of `D`, and the immediate transitions in `A` and `B` all occur at the same simulation time. They are separated by delta delay to place an explicit order on the three events within the simulation. At some time later in the simulation, the falling edge of `D` begins the next series of transactions. These three events all occur at the same simulation time, but the simulator separates them by delta delay to maintain the ordering.

3.3.2 Elaboration The end result of the elaboration of a model is the creation of a set of processes and their interconnections (13:12–1). These processes may then be executed to simulate the model. An interconnection between process A and process B means that process B uses the value of a signal that is computed by process A. These interconnections show the dependence of one process on another. The dependence information constrains the order in which processes must be executed during simulation.

Elaboration is defined for design hierarchies, declarative parts, statement parts, and

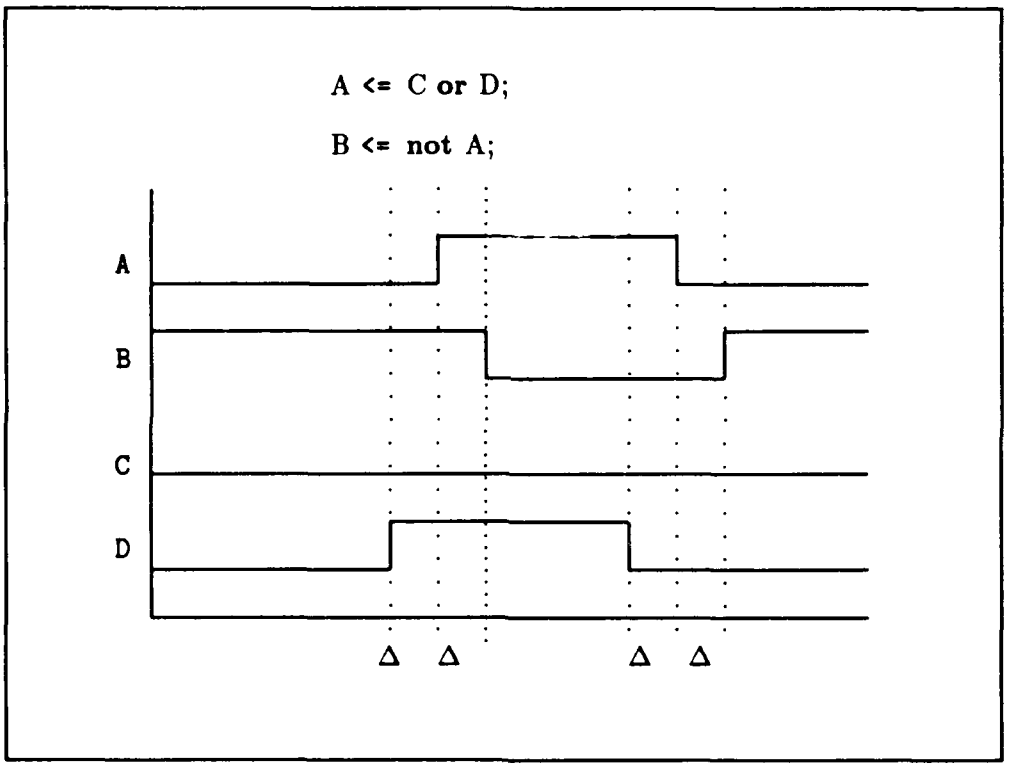

Figure 3.6. Delta Delay

concurrent statements within a VHDL model (13:12–1). Elaboration may be viewed as a recursive process. Before a unit may be elaborated, any not yet elaborated sub-units must be elaborated. In this way, the overall design hierarchy is elaborated. Elaboration of the declarative part of a unit (for example, a package or subprogram) eventually results in the creation of the objects declared within the unit. These objects may be signals, variables, records or arrays. This requires the elaboration of the type, subprogram, and subtype declarations within the declarative part (13:12–4).

Elaboration of the statement part of a unit results in the creation of a process for each process or signal assignment statement (13:12–8). This creates the *driver* which is responsible for maintaining a signal's projected output waveform (13:12–5). The initial value of each signal driven by the process is entered into the projected waveform during elaboration.

3.3.3 Execution Once the model has been elaborated, it may then be executed. The IEEE standard describes a kernel that coordinates the activity of the user-defined processes during the simulation. The functions of this kernel must be distributed in a parallel simulation, otherwise the single process will become the bottleneck. The functions of the kernel are to:

- propagate signal values and update implicit signals
- detect events and schedule processes to run in response to the events
- and, maintain the current value of all signals, including implicit signals (from **guard** statements) and attributes, such as **STABLE** and **QUIET** (13:12–9).

The VHDL simulation cycle repeatedly advances the simulation clock and updates both the explicit and implicit signals. The exact steps in the cycle will be defined in Section 3.3.5. A port is a special signal declared in an entity declaration or an interface list of a component declaration. A port is treated much like a normal signal, except for the presence of the mode declaration constraining the direction of data flow allowed through the port.

3.3.4 Propagation of Signals The kernel must maintain the current value of every signal and also the drivers for each signal. Each process that assigns a value to a signal contains a driver for that signal. A driver contains a signal's projected output waveform. The projected output waveform is a sequence of transactions on a signal ordered by their time of occurrence. Each transaction has two components, a value component and a time component (13:9–4). As simulation time advances, the transactions in the projected output waveform of a given signal will, in succession, become the value of the driver (13:12–9). The transaction whose time component is not greater than the current simulation time determines the value of the driver. The signal's value is a function of the current values of its drivers.

When a driver acquires a new value by advancing simulation time and using the transactions in the projected output waveform, the driver is said to be *active*. A signal is *active* if one of its sources or subelements is active, if the signal is a formal part in a port association list and the corresponding actual signal is active, or if the signal is a subelement of a resolved signal which is active (13:12–10).

As part of updating a signal during one iteration of the simulation cycle, the kernel determines two values for each signal. The *driving* value is the value provided as a source to other signals, and the *effective* value is the value obtained by evaluating the signal in an expression (13:12–10). These two values are not always the same. The driving value of a signal S is determined using the following rules:

- If S has no source, the the driving value of S is the default value associated with S.
- If S has one source that is a driver, and S is not a resolved signal, then the driving value of S is the value of that driver.
- If S has one source that is a port, and S is not a resolved signal, then the driving value of S is the driving value of the formal part of the association element that associates S with that port. The driving value of the formal part is obtained by evaluating the formal part, using the value of the signal denoted by the formal designator in place of the formal designator.

- If S is a resolved signal, then the driving value of S is the same as the resolved value of S obtained by executing the resolution function associated with S, where that function is called with an input parameter consisting of the concatenation of the driving value of the sources of S. (13:12–10)

In short, the driving value of S is the value of its driver passed through a resolution (type-conversion) function if necessary. In the case of a port, the actual parameter is evaluated in place of each formal parameter contained in the port declaration. Composite types, such as arrays and records, are treated as an aggregation of scalar types evaluated using the above rules (13:12–10). This method is recursive when ports are considered. For example, if S appears in a port association list, then the driving value of the corresponding actual port must be determined before the driving value of S can be determined.

Once the driving value is determined, the effective value of S can be computed using these rules:

- If S is a signal declared by a signal declaration, a port of mode **buffer**, or an unconnected port of mode **inout**, then the effective value of S is the same as the driving value of S.
- If S is a connected port of mode **in** or **out**, then the effective value of S is the same as the effective value of the actual part of the association element that associates an actual with S. The effective value of an actual part is obtained by evaluating the actual part, using the effective value of the signal denoted by the actual designator in place of the actual designator.
- If S is an unconnected port of mode **in**, the effective value of S is given by the default value associated with S. (13:12–11)

As before, composite types are treated as aggregations of scalar types subject to the above rules. This method is also recursive. For example, if S is a formal port of mode **in**, the driving value of the corresponding actual parameter must be evaluated before S can be determined.

The effective value of a signal is used to update the value of the signal during any simulation cycle. The rules for updating a normal signal S are:

- If S is a signal of some type that is not an array type, the effective value of S is used to update the current value of S. A check is made that the effective value of S belongs to the subtype of S, and the effective value of S is assigned to the variable representing the current value of the signal.
- If S is an array type, the effective value of S is implicitly converted to the subtype of S. The subtype conversion checks that for each element of S there is a matching element in the effective value, and vice versa. The result of this subtype conversion is then assigned to the variable representing the current value of S. (13:12-11)

Whenever updating a signal causes its value to change, an *event* is said to have occurred on the signal (13:12-11). This event may cause processes to execute in the current simulation cycle. The implicit signals `S'STABLE(T)`, `S'QUIET(T)`, and `S'TRANSACTION(T)` have slightly different rules.

- The implicit signal `STABLE` is updated if and only if an event has occurred on S or the driver of `S'STABLE` is active. If an event has occurred on S, then `S'STABLE(T)` is assigned the value FALSE. If the driver is active, then its value is assigned to `S'STABLE(T)`
- The implicit signal `QUIET` is updated if and only if S is active or the driver of `S'QUIET(T)` is active. If S is active, then `S'QUIET(T)` is assigned the value FALSE.

3.3.5 The VHDL Simulation Cycle Execution of a VHDL model consists of an initialization phase followed by repeated execution of process statements in the model. During each cycle, the values of all signals in the description are computed. If an event occurs on a signal, the process statements sensitive to that signal will resume and be executed as part of the simulation cycle (13:12-13). The current simulation time at the start of simulation is assumed to be 0 nanoseconds. The initialization phase of execution consists of the following steps:

- The driving value and the effective value of each explicit signal are computed, and the current value is set to the effective value.
- The value of the implicit signals `S'STABLE(T)`, or `S'QUIET(T)` is set to True.
- The value of each implicit guard signal is computed by evaluating the guard expression.
- Each process in the model is executed until it suspends. (13:12-14)

Each simulation cycle consists of five steps:

1. If no driver is active, then simulation time advances to the next time at which a driver becomes active or a process resumes.
2. Each active explicit signal in the model is updated. This may cause events to occur.
3. Each implicit signal in the model is updated. This may cause events to occur.
4. For each process P, if P is currently sensitive to a signal S, and an event has occurred on S *in this simulation cycle, then P resumes execution.*
5. Each process that has just resumed is executed until it suspends. (13:12-14)

The event-driven nature of VHDL simulation is reflected in rule 1. This rule allows a simulator to skip over time periods where no events (signal changes or process activations) occur.

3.3.6 Summary The simulation features of VHDL provide for an easy way to test new designs. The requirements for VHDL simulation describe an event-driven simulation kernel which coordinates the actions of the individual VHDL processes.

IV. Design of Distributed Simulation Kernel

This chapter introduces the overall structure and function of the distributed VHDL kernel. The VHDL model is divided into physical and logical processes in order to use the Chandy-Misra model of distributed simulation. To provide for quick implementation, the kernel is hosted on University of Virginia's Spectrum simulation testbed (26).

The IEEE standard for VHDL defines the purpose of the *kernel process* as follows.

> The kernel process causes the execution of I/O operations, the propagation of signal values, and the updating of value of implicit signals; in addition, (it) detects events that occur and causes the appropriate processes to execute in response to those events. (13:B-8)

The design of the distributed kernel must accomplish the same purpose. The functionality of the kernel process is distributed through all of the processes and processors in a distributed simulation.

4.1 Characteristics of VHDL Simulation

The design of the VHDL kernel must support a very wide variety of simulations. When Reynold's classifications (25) (see Section 2.4) are applied to VHDL simulation, a wide range of results is possible. Many of the characteristics of VHDL simulation depend on the design of the circuit under simulation. Table 4.1 lists Reynold's classifications and how VHDL simulation fits in each category. Most of the categories are model-dependent, which means the circuit under simulation determines the value for the characteristic. For example, a 'normal' circuit simulation will be deterministic, but if race conditions exist in the design, the simulation will likewise be nondeterministic. The connectivity of a VHDL simulation is also directly related to the connectivity of the circuit being simulated.

There are two characteristics that hold for all VHDL simulations. No queueing occurs in VHDL simulation, since a process is activated immediately whenever an event occurs on a signal in its sensitivity list. As explained in Section 3.3.1, a logical process will also

Table 4.1. Characteristics of VHDL Simulation

Reynold's Dimension	VHDL Property
Determinism	Model-Dependent
Queueing	Absent
Processing Delays	Present
Causality	Consumptive
Balance	Model-Dependent
Activity Level	Model-Dependent
Connectivity	Model-Dependent

have a processing delay of a least a 'delta-delay' unit. Normally, there will some processing delay related to the propagation delay of the circuit being simulated.

The most important characteristic of VHDL simulation is its consumptive nature. The consumption of input events (where an event represents a change in a signal's value) is required by the IEEE standard (see Section 3.3.4). If a signal is re-evaluated and its value is unchanged, then no output event is generated, thus input events are consumed. It is critical that the distributed simulation kernel avoid deadlock in this situation.

4.2 Physical Processes

The VHDL language is well-suited to distributed simulation and especially to the Chandy-Misra model of physical processes. The Chandy-Misra technique models concurrent physical processes that occur in a system under simulation using logical processes that exist only for the computer simulation of the physical system. In extensions to the original Chandy-Misra technique, a logical process may simulate more than one physical process (18).

The VHDL standard is written in terms of concurrent processes (13:12–1), where each process represents a concurrent statement. Processes in the simulation may result from a signal assignment statement, a component instantiation statement composed of many sub-processes, or a large block of sequential code embedded in a concurrent process statement. All of these various types of processes operate alike and will be called *VHDL*

```
entity c100 is
end c100;
architecture example of c100 is
    signal A, B, C;
begin
    A <= not B and C after 10 ns;
    B <= A xor C after 15 ns;
end example;
```

Figure 4.1. Example of VHDL Processes

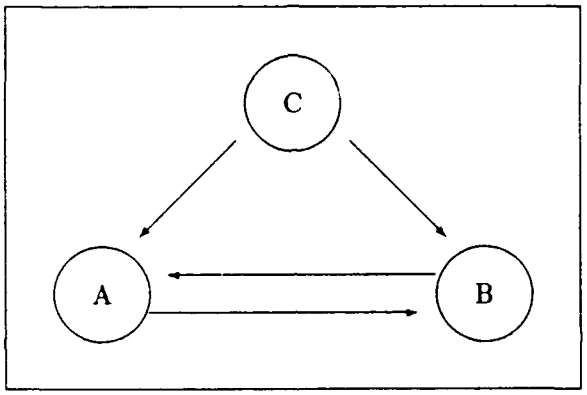

Figure 4.2. Logical Process Structure for Figure 4.1

processes throughout the remainder of this report. The elaboration of a VHDL model creates a collection of VHDL processes interconnected by nets (13:12–1). These VHDL processes form the physical processes of the Chandy-Misra simulation model, and the interconnection net defines the connections that determine the message channels needed. Each physical process is modeled by a logical process in the VHDL simulation. For example, the VHDL model in Figure 4.1 contains two VHDL processes, one for each concurrent signal assignment statement. If we assume, for the moment, an additional process for C, this arrangement can be shown in Figure 4.2 as three circles representing the signals A, B, and C. The arrows show the sensitivity of signals to one another and also the interconnection nets between processes. The signal at the head of the arrow is sensitive to changes in the value of the signal at the tail of the arrow.

4.3 Actions of a Logical Process

The VHDL simulation is divided into logical processes, where each logical process directly corresponds to a VHDL process as determined by the VHDL standard. Each logical process provides the driver (see Section 3.3.4) for one or more signals in the VHDL model. Again using Figure 4.1 as an example; one logical process is required to provide the driver for signal A, and another logical process provides the driver for B. Therefore, there are two logical processes in the simulation and whenever the new value of a signal is computed, it must be sent to any processes that use that signal. The process using the value of a signal is said to be sensitive to that signal, and each process has a *sensitivity list* of signals that it uses.

Since this abstraction requires a logical process for each signal, Figure 4.2 may be interpreted as showing the logical processes and their interconnections instead of signals. The sensitivity list of a process corresponds to the arcs leading towards a process in Figure 4.2. The simulation kernel maintains the sensitivity list of each process in a way that allows determining the 'inverse' of the sensitivity list, called the *dependency list*. The dependency list tells a logical process which other processes must be notified when a signal changes value. The dependency list corresponds to the arcs leading away from a process in Figure 4.2. Of course, if the process controlling signal A is sensitive to the value of signal B, then the process controlling B is dependent on signal A.

A *VHDL event* occurs on a signal only when a signal changes value, not every time it is recomputed or the simulation clock advanced (13:12–11). This reduces the message traffic in a distributed simulation and process activations in a sequential simulation. Each VHDL event creates a new transaction that is attached to the signal's driver, as required by the IEEE standard. For example, whenever A is recomputed and the value of A changes, the logical process responsible for computing B must be notified so that the value of B may be updated. This notification takes the form of a simulation event that is processed using the underlying Spectrum simulation primitives (*Send Event, Receive Event, Advance Clock*, etc.). The distributed simulation kernel will handle the simulation event by sending a message to every process that needs to know the new value and by sending a null message to all of the other processes in the dependency list (as discussed in Section 2.3.2.1). Since

the new value of a signal must often go to *all* of the processes in the dependency list, many of the null messages required by the Chandy-Misra algorithm can be eliminated. The event also activates the receiving VHDL process if the synchronization constraints (on advancing the clock) of the Chandy-Misra algorithm have been satisfied. The simulation event must contain the information about the VHDL transaction that it represents. The event message contains the name of the signal that changed, the new value of the signal, and the simulation time at which the change occurred. This gives the recipient process enough information to advance its simulation clock and compute a new value for its output signal.

As noted by Soule and Gupta (28), a special problem can occur when simulating logic circuits. It is possible for a logical process to accept an input event (signal change) and to not propagate that event to its dependent processes. For example, if a logical process is simulating a simple Boolean OR gate, and one input is 0 while the other input is 1, then the output must be a 1 as well. If the 0 input now changes to a 1, the output remains 1 and no VHDL event is generated. If the logical process modeling this gate were part of a loop of logical processes which depended on each other, then deadlock would occur since this event would be prevented from propagating around the loop of processes. For an arbitrary VHDL simulation to avoid deadlock, the synchronization technique must ensure that events which are consumed by a logical process do not cause deadlock. In order to prevent deadlock, a null message must be generated and sent to all of the processes sensitive to the current process to update their in-coming channel clocks. The initial design for the VHDL kernel had null messages generated by the application to prevent deadlock in this situation. This approach eliminates many of the advantages of Spectrum, since the application depends on the existance of null messages to be deadlock free. If another synchronization (filter) module was selected, the application-generated null messages may no longer work. Spectrum's filter module for VHDL was changed to generate null messages at the apropriate points to eliminate the need for application level null messages. These modifications are discussed in Chapter V, and the Spectrum simulation testbed (see Section 4.7) now handles events that are consumed.

The basic actions of a logical process can be shown in Figure 4.3, which shows the

```
Algorithm IV.1
while simulation is not done do
    wait for event from another process
    compute new output value
    if value has changed then
        send new value to dependent processes
    end if
end while
```

Figure 4.3. Simplified Logical Process Algorithm

```
process
    wait on (Clk) until Clk = '0';
    Q <= D after 10 ns;
    NQ <= not D after 10 ns;
end process;
```

Figure 4.4. Example process with a **wait** statement

simplified logical process algorithm. In short, a logical process waits for changes in the values of signals in its sensitivity list. When an input signal changes, the VHDL process is activated and an output value is recomputed. If the output value has changed, a new transaction is created, and it is sent to all of the processes which are sensitive to it.

There is one case that isn't handled by Algorithm IV.1. This case involves a process block with a **wait** statement, as shown in Figure 4.4. This example implements a 'D' flip-flop, which can only change the outputs **Q** and **NQ** when the clock signal is low. When a process block has a **wait** statement, it cannot execute its sequential code (signal assignments in this case) until the wait-condition is satisfied. Therefore, the main simulation loop must be slightly modified, as shown in Figure 4.5, which contains Algorithm IV.2. The modification requires determining if the wait-condition is satisfied, and proceeding with the simulation loop only if the wait-condition is satisfied. If the wait-condition is not satisfied, then the logical process must wait for another event. For concurrent signal assignment statements and **process** blocks without **wait** statements, the wait-condition is always satisfied, and the action of the logical process is identical to the simplified algorithm

```
Algorithm IV.2
while simulation is not done do
    wait for event from process
    if wait condition is satisfied then
        compute new output value
        if value has changed then
            send new value to dependent processes
        end if
    end if
end while
```

Figure 4.5. Logical Process Algorithm

(Algorithm IV.1).

The example process block in Figure 4.4 has an additional complication. The process block drives two signals (`Q` and `NQ`). Since the signal assignments within a process block are sequential signal assignment statements, the two assignments must be made by a single logical process in the simulation. This logical process is sensitive to both the `Clk` and `D` signals. This is handled by using Algorithm IV.2 and computing new output values for both signals `Q` and `NQ`.

This generalization allows a logical process to drive more than one signal. The logical process' sensitivity list is the union of the sensitivity lists of the internal signal assignment statements. Each process has a list of signals that it drives, and another list of signals to which it is sensitive. This allows the simulation kernel to properly schedule processes to execute.

4.4 Input-Vector File Processing

There is no explicit driver for signal `C` in the example of Figure 4.1, so it is handled differently by the simulation. Since there is no explicit driver for `C`, it will never change value when simulated. One of the useful features of a VHDL simulation is the ability to provide values for signals at any moment in simulation time by means of an "input-vector." The input-vector technique allows the designer to give values for signals at precise

```
Algorithm IV.3
read first vector input time
while simulation is not done do
    if vector input time = ∞ then
        Broadcast end of simulation messages
        Terminate
    else
        read new values from vector file
        Broadcast the new values to other processes
        read next vector input time
    end if
end while
```

Figure 4.6. Algorithm for the Input-Vector Manager Process

simulation times, and these values are treated as normally occurring transactions by the remaining processes in the simulation. In the sequential AFIT simulation kernel, the input-vector is handled by a file which is read at the appropriate moment to update the simulation kernel's value for signals listed in the file. In a distributed approach, the input-vector file is handled by a single process that is responsible for reading the file and updating the signals listed in it. This process is called the input-vector manager, and is an additional process in the distributed simulation that doesn't appear in the original VHDL model.

In Figure 4.2, the circles represent the signals in the VHDL model. Since each signal also corresponds to a logical process, signal C must have a process to control its value. To correctly model the situation, the process responsible for signal C must be the input-vector manager process. The input-vector manager may handle more than one signal, since it must compute the values for all signals without VHDL processes. The input-vector manager consults the vector file for the next simulation time to update the signal values. The new values of the signals are read and broadcast to the logical processes that are sensitive to the signals. This allows the designer to control the value of C externally through the vector input file and to test the operation of the circuit. The basic operation of the input-vector manager is summarized in Figure 4.6 as Algorithm IV.3.

One of the limitations of this distributed design is that the input-vector file cannot

change a signal for which a VHDL process exists. This is only a limitation in comparison with the existing sequential simulator, since the IEEE standard doesn't require a input-vector capability. In the sequential simulator, the input-vector file can change any signal since the sequential kernel has complete knowledge of all signal values. In the distributed simulation, the signal values (and sensitivity lists) are partitioned among the logical processes, so that only a single process may update a particular signal. This process can either be a logical process modeling a VHDL process, or it may be the input-vector manager process.

4.5 Transaction Logging

The goal of VHDL simulation is to be able to trace the changes in signal values as simulation time advances. This requires keeping track of all of the transactions and events that occur on signals. To be able to recover the activity on a signal, all of the transactions are logged in a file. Transactions that represent changes in signal values (events) are tagged in the log file. To help sort out the transactions, each logical process (responsible for a different signal) maintains a separate log file. The log files allow the reconstruction of the values of all signals in the VHDL model at any moment in simulation time.

4.6 Build Program Interface

The BUILD program is responsible for constructing a large part of a VHDL simulation. The build program constructs the model-dependent portions of the simulation program, while the kernel provides the model-independent capabilities. The model-dependent portion of the simulation must be produced by the VHDL analyzer and simulation build phase working together. In the AFIT implementation, these two phases compile the VHDL source code into routines in the C programming language. Each VHDL process is simulated by a single C routine. For processes with **wait** statements, an additional routine must be constructed that checks the wait-condition to allow the process to proceed. Building a simulation of a VHDL model is much like compiling a program. The analyzer performs the initial semantic analysis, while the build program is the code-generation phase of the VHDL 'compiler.'

The translation of VHDL to C code within a **process** or signal assignment statement is relatively straight forward. The mechanism used by Pompilio (21) to build C functions for the serial simulator can be used again for the distributed simulator. Each signal of type BIT or BOOLEAN can be represented by a C **char** variable. The VHDL **and**, **or**, and **not** operators map directly to C's Boolean operators (**&&**, **||**, and **!**). Using these conventions, the build program can construct a C routine that computes the future value of a signal (or multiple signals as in Figure 4.4) and constructs a new transaction to be placed in the signal's driver.

There are two important tables that are used throughout the kernel. These are the signal and process tables. The *signal table* maintains all of the data for each signal in the model, including the current value, the drivers and any transactions. The *process table* maintains the sensitivity and dependency lists for each process, as well as the list of signals driven by each process. The build program must ensure that these tables are initialized properly in order for the kernel to execute properly. The exact layout of the signal and process tables is described in Chapter V.

4.7 Spectrum Simulation Testbed

Paul Reynolds at the University of Virginia has developed the Spectrum Parallel Simulation Testbed (26) for use in exploring techniques for distributed simulation. Spectrum provides a way to develop portable simulation applications that can run using a variety of simulation control methods and on different underlying hardware. Spectrum provides simulation primitives such as *Get Event*, *Post Event*, and *Advance Time* that provide a machine- and control method-independent interface for simulation applications such as VHDL. The control method, such as Chandy-Misra or Time Warp, is implemented inside Spectrum through filters that change Spectrum's behavior. The filters for Chandy-Misra processes handle the generation, transmission, and receipt of null messages. The application never has to handle a null message or worry about deadlock. The constraints on advancing the simulation clock (due to input channel clocks) are handled within the filters as well. If a logical process has not satisfied the Chandy-Misra constraints, the filters will block the process and wait for another message (either an event or null message)

before proceeding. Spectrum also requires a machine-dependent layer that provides basic message passing services. Porting Spectrum to a new environment requires implementing only the message-passing services. AFIT has supported the development of Spectrum by porting the system to BSD Unix systems and the Intel iPSC/2 Hypercube. The Unix implementation runs on a single computer as a set of cooperating processes using standard Unix interprocess communications, while the Hypercube version is a true parallel system using any number of Hypercube processors. Spectrum supports a scalable design where the number of logical processes is not limited by the number of processing nodes available. In addition, the assignment of logical processes to computing nodes is made for each run, and can be varied to suit the individual application or input data set.

The VHDL simulation kernel is designed as a Spectrum application to facilitate quick implementation and provide portability to new environments. Spectrum is based on logical processes that can operate like Chandy-Misra logical processes when the appropriate filters are used. A slightly modified version of the Chandy-Misra filters are used for the VHDL kernel. The modifications include eliminating redundant null messages before transmission, and maintaining the VHDL simulation time and delta (see Section 3.3.1) independently of the Spectrum clock time. The mapping of physical processes to logical processes described in the previous chapter actually maps VHDL processes onto Spectrum processes in this application of distributed simulation.

4.8 Summary

A complete VHDL simulation will have one input-vector manager process and one additional logical process for each concurrent signal assignment or **process** statement. This partitioning of the VHDL model into processes is defined by the IEEE standard. The model generation or build phase of the VHDL process is responsible for identifying and constructing the C routines that simulate each VHDL process. The simulation kernel is responsible for propagating signal values between processes and coordinating advances in the simulation clocks using the Spectrum simulation testbed.

V. Distributed Implementation

This chapter describes the implementation of the distributed VHDL simulation kernel. This implementation runs over the Spectrum distributed simulation testbed, and is portable between the iPSC/2 Hypercube and BSD Unix systems. This chapter is divided into four parts. The first part describes the interface between the build-program and the kernel; the second part describes some of the internal workings of the kernel. The required initialization of data structure is described next, and finally the restrictions and limitations of this distributed kernel are discussed.

Throughout this chapter, examples for the data tables and simulation routines are taken from a simulation of a 3-bit counter constructed from D flip-flops. The original VHDL source code for this model is shown in Figure 5.1. This is not the best way to model a counter, since it doesn't make use of component instantiation, but it does give a good example of many features of the simulation kernel. The complete counter is composed of 6 VHDL processes. Each bit of the counter is represented by a pair of processes—one for computing the input to the flip-flop, and one for computing the normal and complement output signals of the flip-flop. The least significant bit of the counter is modeled by the third process block, which computes `Q0` and `NQ0`. This process is sensitive to signal `D0`, since it appears on the right-hand side of the signal assignments. Also, this process is sensitive to the `Clk` signal, since it appears in the **`wait`** statement.

5.1 The Intel iPSC/2 Hypercube

The goal of this thesis is an Intel iPSC/2 Hypercube implementation of a VHDL simulation. The Hypercube features between 8 and 128 computing nodes, each composed of an Intel 80386 processor, four to sixteen megabytes of memory, and an optional Wietek 1167 floating point coprocessor. The iPSC/2 is a distributed system where processors cannot share memory; they can only communicate via messages exchanged between nodes. The processors in the iPSC/2 are connected using a cube-connected network (29:194). The logical process structure of the Chandy-Misra technique and this VHDL implementation

```
entity counter is
end counter;

architecture dataflow of counter is

     signal Clk :  bit;
     signal q2, q1, q0 :  BOOLEAN;
     signal nq2, nq1, nq0 :  BOOLEAN;
     signal d2, d1, d0 :  BOOLEAN;

begin

     d2 <= (q2 and nq1) or (nq2 and q1 and q0) or (q2 and q1 and nq0)
           after 5 ns;
     d1 <= (nq1 and q0) or (q1 and nq0) after 5 ns;
     d0 <= nq0 after 5 ns;

     process
     begin
           wait on (Clk) until Clk = '0';
           q2 <= d2 after 5 ns;
           nq2 <= not d2 after 5 ns;
     end process;

     process
     begin
           wait on (Clk) until Clk = '0';
           q1 <= d1 after 5 ns;
           nq1 <= not d1 after 5 ns;
     end process;

     process
     begin
           wait on (Clk) until Clk = '0';
           q0 <= d0 after 5 ns;
           nq0 <= not d0 after 5 ns;
     end process;
end dataflow;
```

Figure 5.1. VHDL Description of 3-bit Counter

make good use of the message passing architecture. The iPSC/2 at AFIT has eight nodes, each with four megabytes of memory and the numeric coprocessor installed.

Since a large VHDL simulation will be composed of many logical processes, the VHDL implementation can execute multiple logical processes on each computing node. In this fashion, a large simulation can use the entire cube, or smaller simulations may be run using only a portion of the computing nodes available. The assignment of logical processes to nodes is made when the simulation is *loaded.* This allows the same simulation to be run on different size (number of processors) Hypercubes without recompiling. Also, the placement of logical processes onto cube nodes can be varied between runs to measure the effect of communication delays in the Hypercube network.

The language of choice for programming the iPSC/2 is C, due to the support tools available. In addition, the existing build program generates C code for VHDL processes, and the Spectrum software is also written in C. Therefore, C became the language for implementing the VHDL kernel.

5.2 Build Program Interface

The build program is responsible for converting the VIA intermediate form of the VHDL model into compilable C modules that are linked with the simulation kernel. The modules that must be supplied are the individual process simulation routines, wait condition simulation routines if required by the VHDL model, and a routine to initialize the global signal and process tables used in the kernel.

5.2.1 Global Data Structures The two main data structures used in the kernel are the signal table and the process table. These must be initialized for each VHDL model by the code produced from the VIA description. These data structures were taken from the sequential simulator kernel with some small changes. Most of the changes involve additional fields for cross-indexing the signal and process tables.

5.2.1.1 Signal Table The signal table contains an entry for each signal in the model. The structure of each entry in the table is shown in Table 5.1. The table itself

Table 5.1. Signal Table Structure (Signal)

Field Type		Field Name
char	*	signame
int		signumber
int		process_id
int		cur_val
int		last_val
TIME		last_event
TIME		last_trans
int	*	sens_proc
char		datatype
int		lo_bound
int		size
int		element
struct signal	*	next_signal
struct signal	*	mama_signal
struct driver	*	drv_ptr

exists as an array of pointers to **Signal** entries. This array (called **sig_array**) must be sorted on the signal names (**signame**) so that a binary search can be used. In the current implementation, the complete signal table is replicated in each logical process. This is not necessary, however, since only the signal table entries listed in the process' sensitivity list or actually driven by the process are accessed during simulation by a logical process. The remainder of the signal table, which is not used by the logical process, will have out-of-date values in it since it is never updated.

The original layout and choice of data types for this structure is taken from Pompilio's (21) work, with very few changes. The fields in the signal table include the name of the signal, the index of the signal in **sig_array**, the logical process index of the controlling process, the current and previous values, the simulation time of the last event (change in value) and transaction. The data type, low bound, element, size, and parent signal pointers are used only for structured types like arrays and records.

The **sens_proc** field is a pointer to a list of logical processes which are sensitive to this signal. The process table index of the logical process is used in this list. The

Table 5.2. Partial Signal Table for Counter

	Signal Name			
Data Field	Clk	D0	Q0	NQ0
signame	"Clk"	"D0"	"Q0"	"NQ0"
signumber	0	1	8	5
process_id	0	3	6	6
sens_proc	4, 5, 6	6	2, 1	2, 3

Table 5.3. Signal Driver Structure

Field Type	Field Name
Signal *	sig_ptr
Driver *	next_drv
Transaction *	trans_ptr

sens_proc field is used by the kernel to determine which processes to notify of events on a signal. The drv_ptr field points to the driver structure described below. As defined in the IEEE standard, the driver maintains the list of transactions that form the projected output waveform of the signal.

The signal table is referenced during execution to determine the current values of the signals used in a computation.

There are 10 signals in the counter example of Figure 5.1. The signal table has an entry for each signal. The entries for the Clk, D0, Q0, and NQ0 entries are shown in Table 5.2. The signumber and process_id entries are not consecutive because of the other signals in the model. The sens_proc lists indicate that processes 4, 5, and 6 are sensitive to the Clk signal, while only process 6 is sensitive to D0.

5.2.1.2 Drivers and Transactions Table 5.3 shows the layout and types of the driver structure. The driver data structure is composed of three pointers that link the signal structure with the list of transactions for that signal. The VHDL standard allows for the possibility of multiple drivers for a signal, but this is not implemented in either the sequential or distributed kernels.

Table 5.4. Transaction Structure

Field Type		Field Name
Transaction	*	next
TIME		future_time
int		value

Table 5.5. Process Table Entry

Field Type		Field Name
Driver	**	dlist
int		dlist_cnt
Signal	**	slist
int Function		proc_ptr
BOOLEAN Function		check_wait_cond

The **trans_ptr** forms the head of the linked list of transactions. Each transaction is of the form shown in Table 5.4. Since the transactions are used as a linked list, the **next** field points to the remainder of the list. The **future_time** field is the simulation time at which this transaction will take effect, and **value** is the new value the signal will acquire. The IEEE standard requires the time field to be "the relative delay before the value becomes the current value" (13:B–12), but this implementation uses the absolute simulation time in the transaction. This is because the transaction must be sent to other logical processes (which are probably executing at different simulation times) and still be meaningful.

5.2.1.3 Process Table The process table contains an entry for each logical process in the simulation including the input-vector manager process. Since the BUILD program generates process indices from 1 to N, the input-vector manager is process number 0. Each process has an entry with the fields shown in Table 5.5.

The **proc_ptr** and **check_wait_cond** fields of this data structure are pointers to functions. These functions are constructed by the build phase. The **proc_ptr** is the model-specific simulation routine that actually computes new signal values. The **check_wait_cond**

Table 5.6. Partial Process Table for Counter

	Process Number		
Data Field	6	3	0
dlist	**Q0, NQ0**	**D0**	**Clk**
dlist_cnt	2	1	1
slist	**Clk, D0**	**NQ0**	*empty*
proc_ptr	proc6exec	proc3exec	NULL
check_wait_cond	proc6check	NULL	NULL

routine is optional and used to check to see if the wait-condition for a process block is satisfied. The interface for these functions is given in Section 5.2.2.

The **slist** array contains the sensitivity list of the process. There is one entry for each signal used on the right hand side of the corresponding signal assignment statement or sensitivity list of a process block. The **slist** array is used (by the simulation procedure) to look up the current value of each signal when computing a new result.

The **dlist** array is a list of drivers representing the signals modified (or driven) by this process. When the process executes and computes a new result, a transaction is entered into the projected output waveform for the output signal. The simulation kernel broadcasts this transaction to other logical processes to inform them of the change in value (which is schedule to occur at some point in the future). The **dlist_cnt** field is the length of the **dlist** array; it is used within the kernel since the **dlist** must often be scanned.

Table 5.6 shows the process table entries for processes 0, 3, and 6. It shows that process 6 drives signals **Q0** and **NQ0**, and is sensitive to **Clk** and **D0**. The **proc_ptr** field for process 3 and 6 is the name of a procedure for simulating the VHDL process. Process 6 has a wait statement in it, so its **check_wait_cond** field contains the name of a procedure for checking that the wait condition is satisfied. Process 0, which is the input-vector manager process, drives the **Clk** signal and is not sensitive to any signals. Since this is the input-vector manager, the **proc_ptr** and **check_wait_cond** fields are not used.

5.2.2 Build Phase Generated Routines The **proc_ptr** and **check_wait_cond** fields of the process table are pointers to functions. These two functions must be generated

by the build phase. The **proc_ptr** must point to the routine that simulates this logical process, and **check_wait_cond** points to the routine that checks to see if the wait condition (if any) is satisfied. The simulation routine constructed by the build phase must use the following interface (expressed in the C programming language):

```
void
sim_routine(dlist, slist)
Driver *dlist[];
Signal *slist[];
```

The simulation routine is called with two parameters; the **slist** parameter gives access to all of the current signals the process depends on, while the **dlist** parameter gives access to the projected output waveforms for each signal driven by the process.

The counter example of Figure 5.1 contains 6 processes, but they occur in pairs. Each pair models one of the bits of the counter. There are three output processes. each simulating the outputs of the D flip-flops, that model the **process** blocks in Figure 5.1. The output process must drive both the output signal (**Q**) and the complemented output signal (**NQ**), since both of these signals occur in the same VHDL process statement. Figure 5.2 shows the C function for simulating the process statements in the VHDL model. This same function is used for simulating all three process statements; the **slist** and **dlist** parameters controls which signal are affected by executing this function. For example, when simulating the least significant bit of the counter (**Q0**), the value of **dlist[0]** must point to the driver for the **Q0** signal, and the value of **dlist[1]** must point to the driver for the **NQ0** signal. Also, **slist[0]** must point to the signal table entry for **D0**. This allows the routine to use the current value of **D0** in computing its output. Likewise, for the next bit of the counter (**Q1**), **dlist[0]** must point to the driver for **Q1**, and the **slist** list points to the signals used in computing **Q1**. The **post_trans()** function is part of the simulation kernel that enters a newly created transaction into the projected output waveform attached to a driver. The simulation routine in Figure 5.2 creates two transactions, one for each signal assignment occurring within the process statements of Figure 5.1. The three signal assignment statements are modeled by separate processes

```
int
proc4exec(dlist, slist)
        Driver *dlist[];
        Signal *slist[];
{
        Transact *Newtrans();
        extern TIME simtime;
        Transact *newtrans;

        newtrans = Newtrans();
        newtrans->future_time = simtime + 5;
        newtrans->val = slist[1]->cur_val;
        post_trans(dlist[0],newtrans,FALSE);
        newtrans = Newtrans();
        newtrans->future_time = simtime + 5;
        newtrans->val = not( slist[1]->cur_val );
        post_trans(dlist[1],newtrans,FALSE);
}
```

Figure 5.2. Output process for a D flip-flop.

using different simulation routines. Figure 5.3 shows the simulation routine for the DO statement. This simulation process merely creates a new transaction for DO with the value taken from the current value of NQO. As before, the **slist** and **dlist** parameters must point to the proper entries in the signal table. Section 5.4 describes how the simulation kernel and build phase work together to initialize the signal and process tables.

In order to handle VHDL process blocks as described in Section 4.3 for the modified Algorithm IV.2, a routine to check each wait condition is required. This routine returns a Boolean value to indicate whether or not the logical process can proceed with the simulation. If this checking routine returns FALSE, then the simulation routine will not be executed, and the logical process will wait for another incoming event. The interface for the checking routine is very similar to the simulation routine.

```
BOOLEAN
check_routine(slist)
Signal *slist[];
```

```
int
proc3exec(dlist, slist)
        Driver *dlist[];
        Signal *slist[];
{
        Transact *Newtrans();
        extern TIME simtime;
        Transact *newtrans;

        newtrans = Newtrans();
        newtrans->future_time = simtime + 5;
        newtrans->val = slist[0]->cur_val;
        post_trans(dlist[0], newtrans, FALSE);
}
```

Figure 5.3. Input Process for Least Significant Bit.

```
BOOLEAN
proc4check( slist )
     Signal *slist[];
{
     return EQ(slist[0]->cur_val, 0);
}
```

Figure 5.4. Checking routine for **wait**-statement in Figure 5.1

The checking routine can use the **slist** parameter to check whether an expression involving the signals in the sensitivity list is true. An example checking routine for one of the **process** blocks from Figure 5.1 is shown in Figure 5.4.

5.3 Simulation Kernel Internals

The internal structure of the simulator kernel follows from the data structures presented in this chapter and the logical process structure presented in Chapter IV. The main simulation loop directly implements the logical process algorithm (Algorithm IV.2). The Spectrum system provides the routines needed to wait for an event and to send events to other logical processes. Full documentation on Spectrum, including the application inter-

face and internal structure is included in Appendix B. Spectrum also ensures that the constraints required by the Chandy-Misra algorithm are maintained throughout the simulation. The null messages required by the Chandy-Misra algorithm are generated within Spectrum and processed within Spectrum in the receiving logical process. The VHDL kernel never has to worry about generating or processing these null messages. Each event sent between logical processes corresponds to a change in signal value determined by the VHDL kernel. Only transactions that cause a change in a signal's value (i.e VHDL events) are sent to other logical processes.

5.3.1 Null Message Generation Since a logical process in a VHDL simulation is capable of consuming input events (see Section 4.3), Spectrum must ensure that null messages are sent even when an input message has been consumed. The null messages are generated and processed within the filter module of Spectrum. A slightly different approach to sending null messages was used in the customized VHDL filters than was used in the Chandy-Misra filters supplied with Spectrum. The original Chandy-Misra filters sent null messages whenever an event was sent from one logical process to another. These nulls were sent to all other logical processes that were dependent on the logical process. Instead of sending null messages when events are sent, null messages in the VHDL simulation aren't sent until the logical process has finished sending all events in the current cycle and requests its next input event. The output channel clocks maintained by Spectrum are used to determine which dependent processes did not receive a real event from the application. These logical processes are sent a null message to bring the output channel clock up to date. An important side effect of this change is that a logical process may send an event to multiple destinations without generating null messages until all events have been sent. This minimizes the number of null messages as much as possible, and eliminates the need for a special 'event broadcast' capability in Spectrum. Optimization of broadcast capability is necessary for VHDL simulation, since every event is broadcast to all dependent processes. This modification of null message handling in the Chandy-Misra filters is required not just for VHDL simulation, but for any application where events are consumed, and Reynolds (25) has developed another set of extensions to Spectrum to support consumptive processes.

```
signal A, B, C : bit;
A <= B xor C after 12 ns;
```

Figure 5.5. A Problematic VHDL Statement

5.3.2 Transaction Handling When a transaction is generated for a signal, it is entered into the transaction list for that signal's driver. However, this transaction cannot immediately be sent to the logical processes that are sensitive to it. The transaction may possibly be retracted by another incoming event at the same simulation time, and so it cannot be sent to other processes until the clock is advanced. The sample VHDL statement in Figure 5.5 shows how this can occur. If the initial value for signal B is 1, and the initial value for C is 0, then the output value (A) is also a 1. If at some simulation time t an event (with a new value of 0) for B is received, then the output A must change to 0 at time $t + 12$. However, an event may also occur on signal C at time t changing its value to 1. This causes another change in the output signal at time $t + 12$, restoring its value of 1. If the initial change to A at time $t + 12$ had been propagated to its dependent processes, this transaction would cause a glitch in the waveform for A. The semantics of the inertial delay model used in VHDL require that the waveform for A be unaffected by the changes to its input, since the two changes occurring at the same time cancel their effects on A. The VHDL kernel must not send the newly created transaction for A at time $t + 12$ until it can be sure that no other input events will arrive at time t. If an input event arrives at time $t + 1$, then it will not be able to affect the value of A at time $t + 12$, since the signal assignment statement has a delay of 12 ns. Therefore, once the simulation clock has advanced beyond t, either by receiving a VHDL event at some later time, or by receiving null messages from all input process, the pending event on the output signal may be sent to all of the dependent processes. This complication is handled in the `update_current_value` function, which is called whenever the simulation time is advanced.

5.3.3 Simulation Clock Handling The simulation clock for a logical process is maintained in two forms. The VHDL kernel maintains the simulation time and delta increment in separate global variables used throughout the VHDL kernel. Spectrum maintains its

own clock as a single variable internally, and this clock is used as a field in the event structure. Whenever the simulation time and delta must be used by Spectrum, they are packed into a single **long** variable representing the time. This packing process is controlled by macro definitions, so the format can be easily changed if necessary. The macros for packing and unpacking simulation time values are contained in the file **parsim.h**.

5.4 Initialization

There is a large amount of static data that must be entered into the process and signal tables before simulation can begin. The build phase must construct a routine to initialize these tables. The interface for this routine is:

```
void
sim_initialize( sig_array, proc_array )
Signal        *sig_array[];
Process_Entry  proc_array[];
```

This **sim_initialize** routine is responsible for ensuring that the static data properly reflects the logical structure of the model being simulated. The fields that must be initialized in the signal table (**sig_array**) are: **signame**, **signumber**, **process_id**, **sens_proc**, and **drv_ptr**. In order to properly initialize the **drv_ptr** field, a driver structure must be created and its **sig_ptr** field initialized. The driver and signal structures are cross-linked so that the kernel internal routines can access either structure given a pointer to the other.

The process table must also have an entry for each logical process with an additional entry for the input-vector manager process. The **dlist**, **dlist_cnt**, and **slist** fields must also point to the proper signals and driver structures. Each process, except the input-vector manager, must have a model-specific simulation routine to which **proc_ptr** points. The **proc_ptr** field for the input-vector manager process (process 0) is initialized to the NULL pointer.

The final initialization for the simulation kernel is to set the number of logical processes and signals. All of the data structures in the kernel are dynamically sized for the model currently under simulation. Thus, the number of signals and process must be input

to the kernel. This is done using a small file which contains four lines of data. The first line is the name of Spectrum's '.arcs' file (described in Section 5.4.1 and Appendix B), the next line is the number of logical processes in the system, the third line is the number of signals in the model, and the final line contains the number of physical processes. The number of logical processes should match the number of physical processes. The name of this configuration file must be entered on the command line using the '-c' option when starting a simulation run.

5.4.1 Spectrum Initialization Since the VHDL simulation kernel is hosted on the Spectrum testbed, there is some additional set up that must be done. Appendix B contains a detailed discussion of Spectrum's features and services.

The most important initialization information for Spectrum is contained in the '.arcs' file. Spectrum uses the '.arcs' file to determine the logical structure of the simulation. This file is different for each model, since the interconnection network between logical processes changes for each model.

The 'arcs' file has a section for each logical process in the simulation. Each section contains two parts: the input part, and the output part. The input part for logical process x lists the processes that send messages to logical process x. The output part lists the processes to which x sends messages. This duplicates some of the information from the process and signal tables. However, the duplication is necessary to maintain the layered approach of hosting the VHDL kernel on top of the Spectrum simulation testbed. The input processes correspond to the processes listed in the sensitivity list field (`slist`) of the process table, while the output processes correspond to the `sens_proc` field of the signal table. In addition, Spectrum requires a minimum delay for each out-going arc. This delay is the minimum simulation time after an incoming event that a message will be sent on each out-going arc. Normally, this time is equal to the minimum propagation delay (either inertial or transport delay) in the signal assignment statement simulated by the logical process. The delay must be expressed as an encoded simulation time and delta increment value. The format of the encoding is determined by the macros in the file `parsim.h`.

Spectrum uses this 'arcs' file to determine when null messages must be sent, and

also to constrain advances in the local simulation clock. The output delays are used to determine when to generate null messages if an in-coming event doesn't generate an out-going event.

5.5 Implementation Limitations

Although this simulation kernel is designed with the IEEE 1076 standard in mind, it is not a complete implementation. The majority of the limitations can be addressed through proper design of the build phase. The build phase is responsible for mapping the VHDL constructs into logical processes and this kernel supports a very general form of logical process, which will allow a build phase to map almost all VHDL constructs into a logical process.

The simplest problems with the distributed kernel are limitations and restrictions on the types of processes that are supported. The kernel has been tested with data flow descriptions composed of concurrent signal assignment statements. Also, process blocks have been tested to ensure the wait-condition mechanism works. Further testing is required to support the wide variety of signal types, including user-defined and structured (array and record) types, required in VHDL. Since the build phase can map an array or record into a series of individual `BIT` or `BOOLEAN` signals, support for structured types is nearly complete.

The current design of the wait-condition checking assumes that the wait condition is a function of the signal values. The VHDL standard also allows wait-conditions based on time expressions (e.g. `wait for 2 ns`). The interface and structure of the checking routines will need to be re-evaluated to support this construct. In addition, wait statements may occur in the middle of process statements, this construct is not supported in this design.

One limitation with the distributed design is the limit of one logical process being responsible for a signal. The VHDL standard allows multiple drivers (VHDL processes) to contribute to the value of a signal. This capability allows a bidirectional bus structure with multiple devices to be simulated easily. The devices that are not driving the bus can be 'disconnected' through the use of the keyword `null` in a signal assignment statement.

A straight forward mapping of this VHDL structure into logical processes would require multiple logical processes to update the value of the signal representing the bus, or the introduction of a new logical process to handle bus resolution.

5.6 Summary

The distributed VHDL simulation kernel described in this chapter provides a general purpose tool for simulating most VHDL constructs. Simulation of concurrent statements such as the signal assignment statement, and the process statement is supported. The majority of the translation work falls to the build phase, which is responsible for generating the proper logical processes and signals from the VHDL statements used in a model. This kernel provides a flexible and general purpose mechanism for handling IEEE-compliant signal propagation, except as noted in the previous section, and process execution between an arbitrary number of VHDL processes and signals.

VI. Performance of the Distributed Kernel

This chapter measures the performance of the distributed simulation kernel using three test cases. Each test case is timed while computing on two, four, and eight iPSC/2 computing nodes. The VHDL source code for all of these test cases is listed in Appendix A. Some implications of the measurements presented in this chapter are discussed in Chapter VII.

6.1 Measurement Technique

There are two possible methods that can be used to measure the execution time of an application executing on the iPSC/2. The host (cube manager) can measure the time taken for the node processes to complete, or the node processes can time themselves. The system timer available on the host provides the time to the nearest second. The resolution of this timer is too coarse for measuring a VHDL application. Also, it is impossible to get the time for an individual node process using this method. The alternative is to have the node processes time themselves; this provides a way to get the execution time for individual node processes. Fortunately, the iPSC/2 provides a free-running clock on each computing node, with millisecond resolution, which may be sampled by node processes.

Spectrum uses the computing nodes' millisecond clock to measure the execution time of each logical process. The value returned by this measurement is the "wall-clock" time of execution over the total life of the logical process. Although this clock provides millisecond resolution, all execution times in this thesis have been truncated at the hundredth of a second. In the VHDL simulation, this time includes the entire time spent in the 'simulation' routine (including waiting for messages/events), but does not include the time spent in the `sim_initialize` routine.

The values returned by this clock must be interpreted carefully. Since more than one process may be executing on a single computing node, the clock records all of the time accrued by all of the processes sharing the node. For example, assume two processes, A and B, which start at the same time and share node 0. If the wall time for process A is 8.5 seconds and for process B, 10.5 seconds, then because of the multi-tasking occurring

on node 0, the individual execution time (or CPU usage) of process A or B cannot be determined. During the 8.5 seconds that both processes are executing, the usage of the CPU is divided between the processes by the iPSC/2 node operating system. The context switching occurs due to message waits, file input/output, and other system factors beyond the control of an application. The only measurement that can be drawn from these numbers is that the complete system (of two processes) completes its execution in 10.5 seconds. If process A and B are assigned to different computing nodes for execution, then there will be only one process per node and the times reported by the iPSC/2 system will match the execution time (including time spent waiting for messages) of each process.

6.2 Generation of Test Cases

One of the hardest parts of implementing a system like VHDL is finding appropriate test cases to use to validate the simulation kernel. Ideally, the simulation kernel designer will have a library of useful VHDL models with known outputs that can be simulated. This would allow the effects of modifications to the kernel to seen in the results of one or more test cases.

The development of the AFIT environment for VHDL is hindered by the lack of test cases. The subset nature of the AFIT analyzer tool makes it difficult to use an "off-the-shelf" test case. Any large existing model will make use of features (such as separate compilation or design libraries) not yet implemented in the AFIT analyzer. Therefore, the test cases that were used during the design of this simulation kernel are small and somewhat limited in scope.

The test cases that are described in this chapter were developed using a data flow description of some basic components at the gate level. The VHDL descriptions of these models (see Appendix A) will compile using the AFIT analyzer, and, more importantly, the subsequent BUILD phase can be used to generate the majority of the C code necessary to simulate the test cases. Since the BUILD phase generates C code and data structures for the sequential simulator by Pompilio (21), the C code must be modified to initialize the data structures properly for the distributed kernel. Currently, this is done manually, but can be automated by updating the BUILD phase as appropriate.

All of the test cases were simulated using an arbitrary assignment of logical processes to processors. Logical process i was loaded onto processor number i mod c, where c is the number of processors used in the run. This assignment is made when the iPSC/2 is loaded, and can be changed without recompiling the simulation.

6.3 Performance of Some VHDL Test Cases

Each of the VHDL test cases was measured using the free-running node timer described in Section 6.1. This provides the wall-clock time for each logical process. Since the simulation is complete when all logical processes have exited, the maximum of the individual logical process times is considered the execution time of the simulation. Since the input-vector manager process (logical process 0) can be a bottleneck due to its heavy use of disk files, most of the test cases are run without an input-vector file. The input-vector manager still must execute for a very short time to send termination messages. The clock time for the input-vector manager is discarded from the results in this chapter, since it is generally under one second.

The execution times for each logical process in the test cases is contained in Appendix A. The data tables in this chapter contain summarized data consisting of three times for each run. The minimum time records the execution time of the quickest executing logical process, while the maximum time records the execution time of the logical process that finishes last. The average execution time is computed using all of the logical processes (except for the input-vector manager process). The final entry in the data tables is the average number of logical processes per computing node for the system. This number is computed including the input-vector manager and shows the degree of multitasking occurring on each computing node.

6.3.1 The "FREEADD" Test Model The first test case is a free-running adder ('freeadd') simulation. This model is composed entirely of combinational logic. This model simulates an eight-bit ripple carry adder with some additional processes generating the inputs to the adder. This allows the simulation to continue generating events without using a vector-input file. The freeadd model uses 33 logical processes. The only feedback

Table 6.1. Performance of FREEADD Test Case

Execution Time (secs)	Number of Nodes 2	4	8
Minimum	31.09	31.12	31.05
Maximum	53.26	53.11	53.02
Average	41.12	41.07	41.07
Ave # of Tasks	16.50	8.25	4.13

loops in this model occur in the VHDL processes that generate the inputs to the adder; the adder proper doesn't have any feedback loops. The propagation delay as the carry ripples through each bit position increases the time is takes to complete an addition operation. When eight iPSC/2 computing nodes are used for processing, then four logical processes are executing on each node (except for the short time that the input-vector manager process is executing). When only two computing nodes are used sixteen logical processes share a single node. Unfortunately, the iPSC/2 is limited to at most 20 processes per processing node, so this test case cannot be run using a single computing node.

Table 6.1 contains the results for the 'freeadd' simulation. The performance of the distributed simulation is almost exactly the same with two, four, and eight computing nodes. During the life of this simulation, 8624 messages are sent from one logical process to another. This figure includes both event messages (containing an updated signal value) and null messages (which update channel clocks in the receiving process). Messages sent from a process to itself are not included in the 8624 message total, since they don't involve the node operating system.

6.3.2 The "COUNT" Test Model The 'count' model implements a three bit counter using a series of D-type flip-flops. This circuit provides a test of both combinational and sequential logic. This is the same model used in the examples in Chapter V. This model contains only seven logical processes, so it is incapable of using all eight of the available iPSC/2 computing nodes. Table 6.2 presents the results for the count model on two, four, and seven computing nodes. Once again, the performance of this simulation is nearly identical regardless of the number of computing nodes. The time using four nodes (25.92

Table 6.2. Performance of COUNT Test Case

Execution Time (secs)	Number of Nodes 2	4	7
Minimum	27.43	23.99	29.67
Maximum	27.85	25.92	30.25
Average	27.71	25.19	30.11
Ave # of Tasks	3.50	1.75	1.00

seconds) is less than the time on two nodes (27.85 seconds), but the time increases when seven nodes are used (30.25 seconds). Most likely, the seven logical processes in this simulation are unable to keep seven computing nodes busy. The amount of computation involved in computing an out-going event in a VHDL simulation is very low, especially when compared with the time spent waiting for in-coming events. The execution times for the logical process show very little variation. Since this model is so small and generates so many messages, all of the logical processes spend a large portion of their time waiting for messages or processing null messages.

This model is very nearly a worst case model to simulate, and the number of null messages sent is very high. More than 35000 messages (both event and null messages) are processed during this simulation. This system contains many feedback loops, which cause these null messages to be generated. The logical process interconnection structure for the counter is shown in Figure 6.1. There are three feedback loops; the first loop contains processes 6 and 3, the second: 5 and 2, and the third: 4 and 1. As example of how a feedback loop affects the simulation, consider the signals `Q0`, `NQ0`, and `D0`. Appendix A contains the VHDL source code for this test case, and it shows how `Q0`, `NQ0`, and `D0` depend on each other's value. Logical process 3 computes `D0` using `NQ0`, while logical process 6 computes `NQ0` and `Q0` using `D0`. This loop generates lots of null messages to other users of `Q0` and `NQ0`. Logical process 6 sends 16761 messages to its dependent processes, and logical process 1 receives 10714 messages but only sends 2105 messages. The majority of the messages that process 1 receives are null messages which do not always generate output messages. If a null message is still in the input queue waiting to be processed, and another message (either an event or null message) arrives, then the null message in the queue can

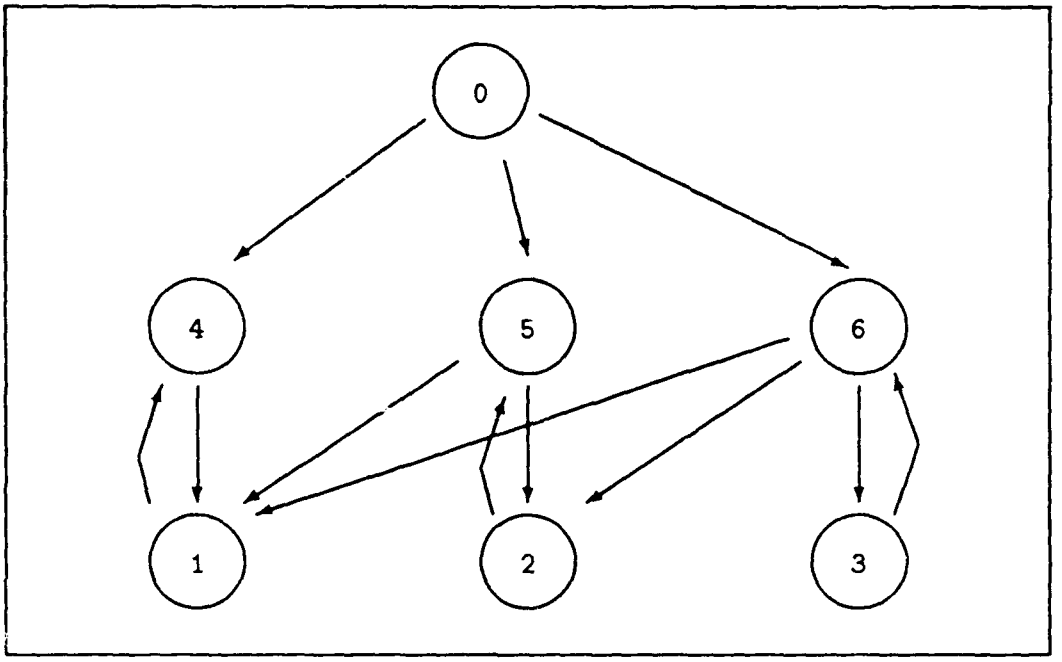

Figure 6.1. Logical Process Structure for COUNT

be eliminated and the later message can take its place.

6.3.3 The "LOOK" Test Model This test case implements a 4-bit binary adder using the look ahead carry approach. The circuit for this model is taken from Preiss' work (22). The adder takes two 4-bit numbers as input, along with the input carry bit, and computes a 4-bit output number and an output carry bit. The 9 input signals are processed by the input-vector manager process. This simulation was run for 3800 ns, and the input signals were changed every 50 ns. Thus, 77 additions were simulated. There are no feedback loops in this circuit. The time from input to output is equal to four gate delays, since the look ahead carry doesn't need to propagate between bit positions. This simulation operates at the gate-level. Each logical process corresponds to a single gate in the circuit and computes a single logic function. There are 37 logical processes in total. Since the input-vector manager process is used heavily in this simulation, its execution-time is included in the summary statistics in Table 6.3. The time for the input-vector

Table 6.3. Performance of LOOK Test Case

Execution Time (secs)	Number of Nodes 2	4	8
Minimum	44.8	44.5	44.5
Maximum	46.8	46.7	46.8
Average	45.5	45.2	45.3
Ave # of Tasks	18.5	9.25	4.6

manager process was neither the lowest nor highest time. From Table 6.3, the maximum execution time for a logical process is nearly identical with 2, 4, or 8 processing nodes. In the LOOK simulation, all of the logical processes execute in nearly the same amount of time. This low variation is is due to the structure of the carry look-ahead adder. The adder is constructed so that only 4 gate delays are required from any input to any output. Since the intermediate carry bits do not have to propagate from the least significant bit to the most significant bit, the logical processes generating the most significant bits do not have to wait for the least significant bits to be computed. In the ripple carry adder used in FREEADD, the most significant bits cannot be computed until the least significant carry bits have been propagated.

6.4 Conclusions

The results of this chapter show that distributed simulation of VHDL is feasible, but the performance of these test cases is very disappointing. There is almost no speed-up gained by parallelizing these examples. Chapter VII presents some reasons for this poor performance, and some specific areas to improve performance.

VII. Conclusions and Recommendations

This chapter summarizes the conclusion drawn from this design and implementation of a distributed simulation kernel for VHDL. Also, this chapter makes some recommendations for further research into simulation of VHDL and general distributed simulation using Spectrum.

7.1 Conclusions

The large number of concurrent constructs in VHDL make distributed "execution" (simulation) of the language very natural. Even though this implementation of one possible VHDL kernel doesn't show speed-up, there are many areas in which the performance can be improved. Other researchers have had more promising results using other variations on the Chandy-Misra algorithm or variants of time warp. Distributed simulation also makes available a much larger memory space for the application's use.

The implementation of this VHDL simulation over the Spectrum testbed worked very well. The services provided by Spectrum did support the requirements of VHDL simulation. In addition, Spectrum provides immediate portability between hosts as claimed. The flexibility provided by Spectrum's use of filters to control the simulation was used to improve the VHDL simulation. The custom VHDL filter module provides complete input and output channel clocks, elimination of redundant null messages, and support for a logical process sending events to itself. These changes have been retro-fitted into the Chandy-Misra module provided with Spectrum to create a "channel clock" module usable with any Spectrum application. One change to the VHDL filter module is specifically for the VHDL kernel. Whenever the simulation time is advanced, the kernel must check for transactions to be sent to other logical processes as described in Section 5.3.2. The support provided by Spectrum through its filters supported the VHDL simulation quite well.

One of the inherent problems in simulating VHDL is the wide variety of simulation models that must be supported. A general purpose simulation kernel will not be able to take advantage of the presence or absence of feedback loops in the simulation. In contrast, a kernel that is tuned to operate without feedback loops will have poor performance when

loops exist in the model. In addition to the differences in connectivity, the models can vary in the amount of processing required when an input event is received. At one extreme, VHDL models can be very much like conventional programs, if large process blocks and behavioral modeling techniques are used. At the other extreme, VHDL models can be large numbers of simple components (acting concurrently) connected in structural models. The structural approach results in a very large number of processes, each of which has very little computing to do when an event arrives. The design tradeoffs for behavioral models are different than the tradeoffs for structural code. The style of VHDL model affects the number and size of concurrent processes that need to be supported by the kernel. The distributed kernel presented here would be best with a relatively few numbers of long, complex logical processes (i.e. behavioral models). Longer processes allow the overhead of waiting for events to be spread over a longer process activation time. However, all of the test cases used in Chapter VI are low level structural models with very simple computational requirements. Thus, the poor performance of this kernel design is not entirely unexpected.

The major factor limiting the performance of this VHDL implementation is the number of extraneous messages processed. Ideally, a simulation running on a small number of computing nodes would use less communications than the same simulation running on a large number of computing nodes. This is not true for this kernel; executing a simulation on fewer computing nodes does not change the number of messages processed by this simulation kernel. Since each logical process is completely independent, null messages are sent between processes residing on the same node. A better implementation would use a single simulation clock for all of the logical processes executing on a given computing node. This would eliminate the need for null messages between processes on the same node, since the only purpose of these messages is to inform other logical processes of the current clock value.

Another approach to solving this problem would be to adopt Mannix's distributed event list algorithm (18) in future kernel designs. This will allow a portion of the set of signals in a VHDL model to be handled by a single process executing on a computing node. Each computing node only will have one process executing, and this process will be responsible for scheduling of the VHDL processes to execute in response to events. This

approach is similar in effect to executing N copies of a sequential VHDL simulation (one on each computing node), with proper synchronization between nodes.

The layered approach used in implementing this VHDL kernel, while providing portability and ease of implementation, does hurt performance slightly. Spectrum provides the basic operations necessary to support distributed simulation. In general, Spectrum could be improved on for any application by implementing a simulation control technique that is tailored for the application. VHDL simulation is no different, and performance could be improved by implementing a custom simulation control algorithm. This simulation control technique could provide the synchronization between VHDL processes at a lower cost both in messages processed and run-time overhead. Spectrum currently needs additional instrumentation to provide event tracking and execution statistics. Spectrum currently does not gather any statistics on the execution characteristics of the logical processes. To gather the statistics in Chapter VI, Spectrum was modified to provide simple counts of the number of messages processed and the time taken by each logical process. However, this is not sufficient; additional statistics that will need to be gathered include the number and types of events sent between various logical processes, and the time spent by a logical process in various states. Some of the states a logical process may be in include: blocked (waiting for a message), executing Spectrum code, executing application code, and initializing Spectrum itself. This instrumentation will be useful for any simulation work with Spectrum.

One of the best sources for improvement in this implementation is to replace Spectrum with a fully custom synchronization kernel within the VHDL kernel. This new kernel will be able to support the broadcast capabilities used by the VHDL kernel using only the minimum amount of synchronization necessary to ensure correct simulation runs. The results achieved by Chawla (7) suggest that some form of time warp algorithm may improve performance versus the Chandy-Misra algorithm used in this thesis.

One of the promising results from this work is the implications about the Intel iPSC/2 performance. The evenness of the execution times, for different numbers of logical processes per computing node, implies that the time for handling *inter*-node messages by the iPSC/2 operating system is the same as handling *intra*-node messages. This fits with the Intel claim

that an application program can't tell (by measuring message transmission delays) that it's running on a cube-connected network. The Intel Direct Connect Hardware routes messages quickly even if it's not 'nearest neighbor' communications.

7.2 Recommendations

There are many directions to continue this research in VHDL and distributed simulation. VHDL is an excellent vehicle for studying distributed simulation, since the language allows the modeling of highly concurrent constructs. The IEEE standard defines the partitioning of a VHDL model into logical processes, making the language further amenable to distributed simulation.

To support more extensive VHDL models using the current kernel, it will be necessary to rewrite the BUILD program to generate code for the distributed kernel. All of the test cases for this thesis were manually reconfigured from the code produced by the sequential version of BUILD, and this process is very tedious and error-prone. The big change in BUILD processing is the code that generates the `sim_initialize` routine; the VHDL process simulation functions work exactly the same in both the sequential and distributed kernels. To be even more useful, the AFIT VHDL analyzer will need to be updated to support more of the VHDL language. The most critical missing features are: separate compilation of entities (components), and support for component design libraries. This will allow the use of large off-the-shelf VHDL models for testing and measuring performance.

In the long term, VHDL simulation tools are going to have to solve the problem of efficiently simulating the wide range of possible VHDL models. As noted before, the number and size of the VHDL processes and other characteristics of a simulation can vary greatly. The automatic generation of programs that efficiently simulate VHDL is going to require heuristics that select the appropriate kernel design based on the size and number of processes that it must support. These heuristics can be developed only after several candidate kernels have been evaluated and the bounds on their performance analyzed. A large suite of test cases, spanning the range of VHDL models, is necessary to test the performance of candidate simulation kernels. The logical process structure used in this thesis seems to work best for models with a small number of large processes, such as

behavioral VHDL code. This hypothesis is currently untested due to the lack of large test cases that conform to the AFIT analyzer subset of VHDL.

An interesting variation on the logical process structure implemented in this thesis is available with shared memory. A multiprocessor system with shared memory, such as the Encore Multimax, can keep the "global" signal and process tables in shared memory. The Multimax can then execute all of the logical processes in the simulation, using semaphores to control access to the signal and process tables. Each logical process normally accesses only a portion of the signal table, allowing concurrent access to different sections of the shared table to proceed without conflict. Unfortunately, the simulation clock would probably become a bottleneck if it were shared, since all logical processes update the clock value.

The DARPA sponsored QUEST project is studying ways to speed up simulation of large VHDL models. The QUEST project is in the process of selecting a standard VHDL analyzer and component library management tools from the commercial vendors. Since the AFIT analyzer supports only a small subset of VHDL, large "off-the-shelf" models could be used in distributed simulation by adopting the QUEST analyzer. The kernel design described in this thesis will be able to be used with the QUEST tools by using a new BUILD phase. The mapping from the QUEST analyzer and library format to the distributed kernel is done using the BUILD phase, so a new BUILD phase should be designed to support as many VHDL features as possible. This BUILD phase would convert the QUEST intermediate form to the logical process structure expected by the distributed simulation kernel. Adoption of the QUEST front-end tools will provide immediate access to the entire VHDL language, as well as the library management tools necessary to build large VHDL test models.

7.3 Summary

The performance results from this distributed VHDL kernel are disappointing due to the mismatch between test cases and kernel design. However, there are many avenues for further research, including improving the distributed synchronization techniques (via Spec-

trum enhancements), and adopting a different kernel design (for example, the distributed event list algorithm).

The Spectrum simulation testbed has shown itself to be robust in allowing the customization of a filter module to support VHDL. There is some room for improvement in instrumentation and broadcast capability.

VHDL simulation depends heavily on the characteristics of the model under test, and development of a high-performance distributed VHDL kernel will depend on matching the kernel's synchronization methods to the model under simulation.

Appendix A. *VHDL Source Code for Test Cases*

This appendix contains a listing of the VHDL models that were used as test cases. The VHDL source code is listed along with the individual execution times for each logical process in the simulation.

A.1 Free-Running, Eight-Bit Adder

This model simulates an eight-bit ripple carry adder for 4000 ns. Some additional VHDL statements are used to periodically update the inputs to the adder, so that an input-vector file is not required. This model uses 33 logical processes in the distributed system. One of these logical processes is the input-vector manager, which terminates after sending termination messages.

```
entity add8 is
end add8;

architecture test of add8 is

        signal y01, y12, y23, y34, y45, y56, y67 : bit;

        signal a0, a1, a2, a3, a4, a5, a6, a7 : bit;    -- input
        signal b0, b1, b2, b3, b4, b5, b6, b7 : bit;    -- input
        signal cin : bit;                -- input carry

        signal c0, c1, c2, c3, c4, c5, c6, c7 : bit;    -- output
        signal cout : bit;               -- carry out

begin

   a0 <= not a0 after 90 ns;
   a1 <= not a1 after 90 ns;
   a2 <= not a2 after 90 ns;
   a3 <= not a3 after 90 ns;
   a4 <= not a4 after 90 ns;
   a5 <= not a5 after 90 ns;
   a6 <= not a6 after 90 ns;
   a7 <= not a7 after 90 ns;
```

```
    b0 <= not b0 after 90 ns;
    b1 <= not b1 after 90 ns;
    b2 <= not b2 after 90 ns;
    b3 <= not b3 after 90 ns;
    b4 <= not b4 after 90 ns;
    b5 <= not b5 after 90 ns;
    b6 <= not b6 after 90 ns;
    b7 <= not b7 after 90 ns;

    c0 <= a0 xor b0 xor cin after 5ns;
    y01 <= (cin and (a0 xor b0)) xor (a0 and b0) after 10ns;

    c1 <= a1 xor b1 xor y01 after 5ns;
    y12 <= (y01 and (a1 xor b1)) xor (a1 and b1) after 10ns;

    c2 <= a2 xor b2 xor y12 after 5ns;
    y23 <= (y12 and (a2 xor b2)) xor (a2 and b2) after 10ns;

    c3 <= a3 xor b3 xor y23 after 5ns;
    y34 <= (y23 and (a3 xor b3)) xor (a3 and b3) after 10ns;

    c4 <= a4 xor b4 xor y34 after 5ns;
    y45 <= (y34 and (a4 xor b4)) xor (a4 and b4) after 10ns;

    c5 <= a5 xor b5 xor y45 after 5ns;
    y56 <= (y45 and (a5 xor b5)) xor (a5 and b5) after 10ns;

    c6 <= a6 xor b6 xor y56 after 5ns;
    y67 <= (y56 and (a6 xor b6)) xor (a6 and b6) after 10ns;

    c7 <= a7 xor b7 xor y67 after 5ns;
    cout <= (y67 and (a7 xor b7)) xor (a7 and b7) after 10ns;

end test;
```

A.2 Three-Bit Counter

This model implements a three-bit counter. It contains both sequential and combinational logic, and tests the **process** statement in VHDL. This model uses only 7 logical processes and executes for a simulated 1000 ns.

Table A.1. Execution Time For FREEADD Simulation

Logical Process	Time Using 2 Nodes	Time Using 4 Nodes	Time Using 8 Nodes
0	2.167	2.164	2.021
1	31.236	31.227	31.135
2	31.227	31.218	31.126
3	31.210	31.212	31.133
4	31.222	31.203	31.125
5	31.196	31.198	31.120
6	31.207	31.188	31.110
7	31.191	31.183	31.118
8	31.162	31.174	31.110
9	31.156	31.168	31.103
10	31.148	31.159	31.095
11	31.131	31.153	31.090
12	31.142	31.144	31.081
13	31.107	31.138	31.074
14	31.100	31.130	31.065
15	31.091	31.123	31.059
16	31.090	31.115	31.052
17	39.556	39.540	39.535
18	39.530	39.532	39.528
19	52.177	52.150	52.109
20	52.149	52.112	52.031
21	52.331	52.263	52.183
22	52.343	52.274	52.193
23	52.495	52.427	52.347
24	52.565	52.458	52.359
25	52.702	52.594	52.512
26	52.732	52.615	52.525
27	52.883	52.766	52.678
28	52.941	52.785	52.708
29	53.071	52.915	52.838
30	53.109	52.955	52.876
31	53.239	53.082	53.004
32	53.263	53.106	53.017

```
entity counter is
end counter;

architecture flowable of counter is

--
--  3 bit counter made from D flip/flops
--
   signal clk : bit;
   signal q2, q1, q0 : BOOLEAN;
   signal nq2, nq1, nq0 : BOOLEAN;
   signal d2, d1, d0 : BOOLEAN;

begin

   d2 <= (q2 and nq1) or (nq2 and q1 and q0) or (q2 and q1 and nq0);
   d1 <= (nq1 and q0) or (q1 and nq0);
   d0 <= nq0;

   process
   begin
      wait on (Clk) until Clk = '0';
      q2 <= d2 after 5 ns;
      nq2 <= not d2 after 5 ns;
   end process;

   process
   begin
      wait on (Clk) until Clk = '0';
      q1 <= d1 after 5 ns;
      nq1 <= not d1 after 5 ns;
   end process;

   process
   begin
      wait on (Clk) until Clk = '0';
      q0 <= d0 after 5 ns;
      nq0 <= not d0 after 5 ns;
   end process;
end flowable;
```

Table A.2. Execution Time For COUNT Simulation

Logical Process	Time Using 2 Nodes	Time Using 4 Nodes	Time Using 7 Nodes
0	0.763	0.790	0.718
1	27.859	25.925	30.256
2	27.846	25.515	30.254
3	27.849	24.000	30.254
4	27.438	25.979	30.128
5	27.448	25.710	30.079
6	27.840	23.991	29.679

A.3 Look Ahead Carry Adder

The final test case is a 4-bit lock ahead carry adder. This circuit is a purely gate-level simulation of an adder. The simulation requires 37 logical processes. The test case ran for 3800 ns and the input signals were updated every 50 ns. Thus, 77 additions were performed. This is the only circuit that uses the input-vector manager process heavily. Table A.3 shows that the time for the input-vector manager is roughly equal to the remaining logical processes. This circuit has the best potential for speed-up since the design doesn't depend on a carry signal propagating between bits.

```
entity lookaheadadder is
end lookaheadadder;
architecture gatelevel of lookaheadadder is

signal t0, t1, t2, t3, t4, t5, t6, t7, t8 : bit;
signal t9, t10, t11, t12, t13, t14, t15, t16 : bit;
signal t17, t18, t19, t20, t21, t22, t23, t24, t25, t26 : bit;
signal t27, t28, t29, t30 : bit;

signal cin, a0, a1, a2, a3 : bit;
signal cout, b0, b1, b2, b3 : bit;
signal c0, c1, c2, c3 : bit;
begin
t0 <= not cin;
t1 <= not (a0 or b0);
t2 <= not (a0 and b0);
t3 <= not (a1 or b1);
```

```
t4 <= not (a1 and b1);
t5 <= not (a2 or b2);
t6 <= not (a2 and b2);
t7 <= not (a3 or b3);
t8 <= not (a3 and b3);

t9 <= not t0;
t10 <= t2 and not t1;
t11 <= t2 and t0;
t12 <= not t1;
t13 <= t4 and not t3;
t14 <= t0 and t2 and t4;
t15 <= t1 and t4;
t16 <= not t3;
t17 <= t6 and not t5;
t18 <= t6 and t4 and t2 and t0;
t19 <= t6 and t1 and t4;
t20 <= t3 and t6;
t21 <= not t5;
t22 <= t8 and not t7;
t23 <= t8 and t6 and t4 and t2 and t0;
t24 <= t8 and t6 and t1 and t4;
t25 <= t8 and t3 and t6;
t26 <= t8 and t5;
t27 <= not t7;

t28 <= not (t12 or t11);
t29 <= not (t16 or t15 or t14);
t30 <= not (t21 or t20 or t19 or t18);
cout <= not (t23 or t24 or t25 or t26 or t27);
c0 <= t9 xor t10;
c1 <= t28 xor t13;
c2 <= t17 xor t29;
c3 <= t22 xor t30;
end gatelevel;
```

Table A.3. Execution Time For LOOK Simulation

Logical Process	Time Using 2 Nodes	Time Using 4 Nodes	Time Using 8 Nodes
0	44.878	44.507	44.505
1	44.982	44.680	44.659
2	45.013	44.791	44.871
3	45.041	44.973	45.199
4	44.889	44.532	44.525
5	44.915	44.550	44.533
6	44.899	44.538	44.530
7	44.933	44.537	44.536
8	44.926	44.544	44.541
9	44.929	44.562	44.552
10	45.344	45.018	44.966
11	45.653	45.605	45.798
12	45.655	45.598	45.791
13	45.383	45.164	45.233
14	44.908	44.550	44.603
15	45.754	45.639	45.821
16	45.457	45.211	45.249
17	44.942	44.570	44.567
18	44.941	44.569	44.562
19	45.807	45.655	45.838
20	45.586	45.252	45.281
21	44.961	44.582	44.571
22	44.953	44.581	44.572
23	44.961	44.586	44.583
24	45.866	45.646	45.861
25	45.668	45.257	45.303
26	44.981	44.601	44.593
27	44.984	44.609	44.597
28	44.980	44.597	44.592
29	46.365	46.260	46.463
30	46.464	46.334	46.516
31	46.423	46.364	46.546
32	46.583	46.318	46.571
33	46.355	46.300	46.503
34	46.728	46.619	46.835
35	46.835	46.700	46.896
36	46.543	46.383	46.568

Appendix B. *The Spectrum Simulation Testbed*

This appendix describes the Spectrum application level interface as used in the VnDL kernel. Spectrum began on a BBN Butterfly computer at the University of Virginia. The Butterfly implementation uses mailboxes in shared-memory to pass messages. As part of this thesis, Spectrum was ported to BSD Unix systems and the iPSC/2 Hypercube. Both of these ports use operating system message passing services instead of shared-memory mailboxes. This appendix describes the version of Spectrum ported to BSD Unix and the Intel iPSC/2 Hypercube and in use at AFIT.

The application should use only the logical process management (Section B.2) routines and the utility (Section B.4) routines. The node level routines (Section B.3) are for the internal use of Spectrum.

B.1 General Structure

The general structure of a Spectrum application consist of a set of C functions that represent each logical process (LP) of the simulation. Each function simulates the LP using the interface routines described below to handle event synchronization. Spectrum requires the communications 'net' connecting the LP's be described in a data file called the 'arcs' file.

B.1.1 Communications Net File This file, generally named `application.arcs`, tells Spectrum which LP's communicate with each other. This file is read during startup, and the arcs information is static throughout a run of Spectrum. Among other uses, this allows the Chandy-Misra filter module to determine when null messages are required, and to whom to send them. Each LP has a section of the file devoted to it. The LPs are numbered from 0 to $N - 1$, and must be listed in this file in order. Figure B.1 gives an example of the format for each section. Only the numbers on the left are part of the data file; however, descriptive comments can be entered at the end of a line following a # character. This makes it easier to keep track of what each line is. This section is repeated for each LP in the system. The lists of LP indices within the section, for example the

```
0                   # LP index
2                   # Number of input LPs
1 2                 # LP indices of input LPs
5 5                 # Polling frequencies of input LPs
0 1                 # Offset of polling frequency
4                   # Number of input lines
1 1 1 2             # LP number for each input line
3                   # Number of output LPs
3 4 5               # LP indices of output LPs
5                   # Number of output lines
3 3 4 4 5           # LP index for each output line
2 2 4 1 7           # Minimum delays for each output line
```

Figure B.1. Example Section From "samp'e.arcs" File

LP numbers of the input or output lines, must also be listed in increasing order. The delay time on output lines is the (simulated) time it takes for an event message to be transmitted from the sending LP to its recipient. The polling frequencies and offsets are not used by the filters implementing the Chandy-Misra algorithm, perhaps this data is required by Reynold's SRADS filters. This example is taken from the comments in the Spectrum source file `lp_man.c`, and an LP may have more than one input or output line connected to a single LP. This capability is not used in the VHDL simulation application.

This information is read into a global data structure before the individual LPs are started. The data structure is declared as `struct lp_info_type` in the file `spectrum.h`.

B.1.2 Main Program Structure The `main()` routine of a Spectrum application sets up a table of functions and arranges to start all of the LP's by calling the routine `lp_level_init()` (see B.2). So that the Spectrum system knows how many LP's are needed, and which C functions simulate each LP, `lp_level_init()` is passed a table listing the functions for each LP. This table also includes a name for each LP that is used to identify the LP at startup. This table is passed to `lp_level_init()` along with an array containing the arguments for each LP (in string form). The exact declaration of this table is contained in the header file `spectrum.h`, which all applications should include.

B.1.3 Logical Process Structure Each logical process should follow this general structure.

1. The first step is to call the routine `lp_init()` to ensure the LP manager is completely initialized.
2. Next is the main simulation loop. This probably will take the general form of waiting for events (with `lp_get_event()`), simulating some activity, and sending new events to other LPs (using `lp_post_event()`).
3. The final activity is to call `lp_terminate()` to shut down Spectrum, and also to explicitly send any terminating messages to other LPs.

B.2 Logical Process Interface

This section describes the routines used in the top-most level of Spectrum. The application uses these routines to send and receive events, and to advance the simulation clock.

```
void
lp_level_init(functions, args)
LP_FUNC_INFO    functions[];
char           *args[];
```

This function is called at startup to initialize the entire Spectrum system. It is responsible for starting each LP, and ensuring that it is able to communicate with the other LP's. For the Sun version of Spectrum, this involves forking new processes and opening sockets for communications. Each LP listens on a socket, that is named with the LP name, for messages from other processes.

```
void
lp_init(who)
int     who;
```

This function should be called at the start of each LP routine to ensure that the LP manager is fully initialized. The who parameter is the logical process index of the calling routine. This routine builds the filter table that determines the filter routines called by Spectrum. It is called after LP startup to allow each LP to use a different set of filters.

```
void
lp_post_event(event)
struct event    *event;
```

This routine is the basic event sending mechanism. The event structure passed to this routine is sent to the proper logical process via the machine dependent communications routines. This routine also calls the appropriate filter routine to allow a filter to eliminate the message.

```
void
lp_post_message(event)
struct event  *event;
```

This routine is used by the Node level (machine dependent code) to notify the LP of new messages arriving from other nodes. It uses `lp_nq_event()` to enter the event into the event queue for the LP. The message filter is called to allow a synchronization protocol to remove a message before a LP sees it.

```
struct event *
lp_get_event()
```

This is the basic routine for blocking a LP until a message arrives. The filter routine is called to allow any arbitrary processing. If no filter is defined, `lp_get_event()` works as follows: If there are no events current in the event list for the LP, then the LP blocks until a message is received from another process. This function returns a pointer to either the first event in the list, or to the event received, and the variable `current_event` is set to the same value. Note, `current_event` is for the use of filter routines, not the application.

```
void
lp_nq_event(event)
struct event    *event;
```

This routine enters an event into the event queue for the current LP. It maintains the queue in time-order. This routine is used by the node-layer code and by some application processes to 'send' messages to themselves. For this to work using the Chandy-Misra filters, the current LP must be listed in the arcs file as an input channel.

```
void
lp_advance_time(new_time)
int             new_time;
```

This routine is used by an LP to advance its notion of simulation time. The parameter `new_time` is the new simulation time, **not** an increment or delta. This routine will call a filter if necessary, so that an synchronization protocol can control the time.

```
void
lp_terminate()
```

This routine is called during termination to partially shut down the Spectrum system; it removes the filters from the filter table. The application is responsible for sending termination messages to other processes and ensuring that the process exits normally.

B.3 Node Level Interface

This section describes the machine-dependent node level. The Logical Process routines (Section B.2) use this layer for initialization and to pass messages. The messages that are passed contain a **struct event** variable.

```
void
node_level_init(functions,args)
LP_FUNC_INFO functions[];
char *args[];
```

This function is called upon startup. It must do whatever is necessary to create the Spectrum environment. In the Unix implementation, this includes setting up the sockets and forking the processes. When this function is called, only a single Unix process exists; this function creates all of the needed Unix processes (1 for each Spectrum logical process). On the Hypercube, this routine receives the logical process routing information from the host program. Since all the Hypercube processes are started by the host program, this function doesn't have to create any processes on the Hypercube. However, for N logical processes, this routine will execute once on the Sun and N times (concurrently) on the Hypercube. This routine is called from `lp_level_init()` in the logical process layer.

```
int
node_receive_pending_messages()
```

This routine processes as many messages as it can, sending each event message to the process with the `lp_post_message()` function. When all the messages have been processed, or if no messages are available, this function returns 0. This means that this function will always return 0, after putting some number of event messages onto the process' queue.

```
void
node_block_til_message()
```

This routine causes the logical process to block until a message becomes available. The message that is received is passed to the process using the `lp_post_message()` function.

```
void
node_send_message(event)
struct event *event;
```

This is the interface to the machine dependent inter-process communication mechanism. The Unix implementation uses sockets, and the Hypercube uses the node operating system provided message passing operations.

```
struct event *
node_create_event(event)
struct event *event;
```

This function creates a new event structure (using **malloc()** to get space), and copies the structure passed to it into the new space. In the original BBN implementation, special action is required to make the event structure in globally shared memory. This is not required on the Hypercube or Unix implementations.

```
void
node_trash_event(event)
struct event *event;
```

This function deletes the event structure passed to it. This reclaims the space and allows it to be re-used. For the Hypercube and Unix implementations, this function simply calls the library **free()** routine. Special action is needed in the original BBN implementation.

```
void
node_delay(delay)
int delay;
```

This function causes the logical process to be put to sleep for *delay* seconds.

```
void
node_report(lpid)
int lpid;
```

This function is called on startup and logs a startup message to the logX file for the LP. The parameter **lpid** is the Spectrum logical process index.

B.4 Utility Functions

Spectrum's utility functions are used internally and may be used by the application to place information in the log files.

```
void
log(inmsg)
char    *inmsg;
```

Each LP has available to it a log file named logX, where X is the LP index. This routine writes the string **inmsg** to the appropriate log file. The string should not have a newline character at the end of a line; one will be appended by the **log** function. Note, there is a lot of tracing information that is placed in the log files by Spectrum itself.

```
void
errlog(inmsg)
char    *inmsg;
```

This function writes the string **inmsg** to the 'errlog' file. The string should not have a newline character at the end of a line; one will be appended by the **errlog** function. This is used for error messages, and other special information that shouldn't be mixed in with the sometimes verbose logX files.

B.5 Modifications Made From Original Version

This section lists the main changes made to Spectrum during use in VHDL simulation. The changes are not very extensive; however, the application must be changed to initialize some global variables that configure Spectrum.

1. Spectrum has been fully ported to the BSD Unix environment and the Intel iPSC/2 Hypercube. This required a host program to load the simulation onto the node processors. In this implementation, the host program loads the simulation n times, where n is the number of logical processes. Thus, the Spectrum initialization code

in the `lp_level_init` and `node_level_init` functions is executed n times, instead of just once.

2. Instead of using the supplied 'ui' program to build Spectrum applications, the Spectrum interface has been changed to use two global variables (INPUT_ARCS and NUM_PROCS). This allows the Spectrum code (lp_man.o, null_mess.o, util.o, CMfilters.o, and a node-layer module) to be placed in an archive file on BSD systems. However, the supplied Makefile just links with the appropriate .o files. The Hypercubes' archive command can't be used (since there's no ranlib command), so this is solved by just linking with the four object modules.

3. The file CMfilters.c contains the definitions that were previously generated by the ui program. There should be a matching file SRADSfilters.c.

4. A new Chandy-Misra NULL message filter module has been written to track input- and output-channel clocks and to avoid sending redundant null messages. A redundant null message is one that doesn't advance the output-channel clock. This filter module is contained in the file `chanclocks.c`. This module works with processes that send events to themselves, if the current process is listed as an input channel. Also, this module doesn't send null messages as part of the **post** filter, but as the first action in the **get** filter. This allows an application to send multiple events, and when Spectrum gets control the proper number of NULL messages is sent.

5. Some minimal resource statistics logging been added for BSD and the iPSC/2. The CPU time taken by each logical process and the number of messages sent or received is written to the log file.

6. The utility functions `log()`, `errlog()`, and `open_file()` have been moved to their own file (util.c).

Note: Most of these changes have only been tested with the null messages filter module. It is possible that the SRADS filter module, supplied with Spectrum, will not work with these changes.

Appendix C. *Programmer's Guide*

The VHDL simulation kernel resides in 23 source files. In addition, there are three header files (`.h files`) for the kernel, and two header files for Spectrum. Each source file (`.c files`), with a few exceptions, contains one C function. The two most important files are `lp-main.c` and `lp-basic.c`; these implement the main simulation loop for regular logical processes and the input vector manager process, respectively. Also, the file `vhdl-null.c` contains the custom Spectrum filters for use in the VHDL kernel.

The VHDL kernel directory contains a Makefile (for the UNIX utility **make**) for compiling the sample VHDL simulations. It will ensure that all of the code is properly compiled, make sure Spectrum is compiled and up-to-date, and build the executable simulation program. The Unix command **make** will compile all five example simulations.

C.1 Header Files

There are five header files used by the simulation kernel.

1. debug.h — This file defines two constants, `DEBUG` and `VERBOSITY`, used to control the amount of diagnostic output placed in the log file. `DEBUG` should be set to 1 or 0, and `VERBOSITY` can range from 1 (least output) to 5 (most output). The diagnostic output is helpful to debug the distributed simulation kernel, and requires an understanding of the internal structure of the kernel.
2. mathlib.h — This file contains macros for arithmetic and logical operations on signals. The build-generated C code uses these macros to evaluate expressions.
3. parsim.h — This file defines the Spectrum event types used in the simulation kernel.
4. sim-stru.h — This file defines the structures for the signal and process tables. The definitions for the driver and transaction structures are also included. This file originated with the sequential simulator; some fields were added to the signal table and process table structures, but nothing was taken out.

5. spectrum.h — This file defines the Spectrum event structure. It also defines the return types for the Spectrum routines. The type **struct lp_info_type** is for the use of Spectrum filters.

C.2 Source Files

1. broadcast.c — This file contain a single routine to broadcast messages to all dependent processes. This routine, **BroadcastMessage**, is used by a logical process to send end-of-simulation messages to all dependent processes, and to send empty (or null) messages to all dependent processes.
2. cmdlines.c — This file was taken from the sequential simulator. It contains the routines to parse the command line to select options. The **-c** option is new in the distributed simulator, and this file was changed to read the configuration file described in Chapter V.
3. dump-event.c — This file contains the routine **dump_event**, which is used to print out the contents of a Spectrum event structure. It is used in many of the diagnostic tracing statements.
4. dump-pt.c — This file contains one routine, **dump_process_table**, which prints out the contents of the process table to the error log file on startup. This is used only when the diagnostic trace statements are enabled.
5. dump-sig.c — This file contains another diagnostic routine, **dump_signal_table**, which prints out the contents of the signal table to the errlog log file on startup. This is used only when the diagnostic trace statements are enabled.
6. exp.c — This file contains a single routine, **EXP**, to do integer exponentiation. It is part of the run-time math library.
7. insert-trans.c — This file contains **InsertTrans**, which is responsible for attaching transactions to drivers in accordance with IEEE 1076 requirements. This is one of the most critical routines in the kernel. It is called from both **post_trans** and **post_change**.

8. lp-basic.c — This file contains the main simulation loop for the input vector manager process. This routine is named **lp_vhdl_basic**. It reads the input vector file and sends messages to the other logical processes to update signal values.

9. lp-main.c — This file contains the main simulation loop for each logical process in the simulation. This file contains two routines, **process_signal_event**, and **lp_vhdl_main**. The main loop follows the pseudo-code presented in Chapter IV. The routine **lp_vhdl_main** implements the main simulation loop, and each VHDL event (either an inertial or transport transaction) is processed by **process_signal_event**.

10. mintime.c — This file contains a small routine to determine the minimum of two TIME variables.

11. new-structs.c — This file is taken unchanged from the sequential simulator. It contains routines to dynamically allocate new signal, driver, and transaction structures. Only a portion of these routines are used in the distributed kernel.

12. post-change.c — This file contains a single routine that is used in the input vector manager process. The **post_change** routine is called by **lp_vhdl_basic** when new values for a signal are read from the vector input file. The real work of updating the signal table is done by **InsertTrans**.

13. post-trans.c — This file contains a single routine that forms the entry point into the kernel used by the build-generated simulation routines. Whenever a new transaction is computed for a signal, the **post_trans** routine is called to enter it into the driver for the signal. The actual work is done by the **InsertTrans** function above.

14. read-tra.c — This file is taken unchanged from the sequential simulator. It provides the capability to turn off tracing information for certain signals listed in the 'no-trace' file. This feature is not implemented in the distributed kernel even though this file is compiled in.

15. read-vect.c — This file is taken from the sequential simulator. It contains the routine **read_vector_file**, which is responsible for reading a line from the vector input file. Each line of the vector input file corresponds to a single simulation time and the lists the changes in signal values occurring at that time. This routine is called from

lp_vhdl_basic and uses the post_change function to send event messages, with the new signal values, to dependent logical processes. The code from the sequential simulator attempts to support structured signal types such as arrays and records, but this capability is untested in this implementation.

16. send-read.c — This file contains a single routine, SendReadEvent, which is used by the input vector manager process to send a READ_VECTOR event to itself. The READ_VECTOR event causes the manager process to reawaken and process another line of the input vector file. After processing that line, another READ_VECTOR event is generated to repeat the process.

17. send-trans.c — The send_trans routine in this file is used to generate a Spectrum event from a VHDL transaction structure. Once the event message is created, it is sent to all of the logical processes in the signal sens_proc list. This notifies these processes of the new value of the signal.

18. sfindsig.c — This file is taken unchanged from the sequential simulator. The routine find_signal does a binary search of the signal table and returns a pointer to the signal with a given name.

19. simmain.c — This file contain the main routine of the simulation. It is the startup routine that is first executed when the simulation is run. The main routine calls routines to parse the command line, dynamically allocate the signal and process tables, initialize the tables for both the VHDL kernel and the Spectrum kernel, and finally call the Spectrum function lp_level_init to create all of the logical processes and begin simulation. The lp_level_init function returns only when all logical processes have completed.

20. spect-filt.c — This file contains the build_table routine used by Spectrum to initialize the filter table. The filter table contains five entries: the initialization filter, get event filter, post event filter, post message filter, and time filter. The filter table points to routines in the file vhdl-null.c. This file could be changed to allow different filters to be used for different logical processes.

21. strsave.c — This file contains an additional function for the string library, **strsave**. This routine copies a string into newly malloc'ed space and returns a pointer to the copy. It is used during initialization of the signal and process tables.

22. update-value.c — This file contains a single function, **update_curr_value**, which is called whenever the VHDL simulation time needs to be advanced. When simulation time advances transactions that are attached to a signal's driver may need to be sent to dependent logical processes. These transactions cannot be sent when they are first computed because they may be retracted by additional signal changes occurring at the same simulation time. Therefore, when the simulation clock has advanced, these pending transactions may be committed by writing the transaction to the log file and sending a message to all dependent logical processes.

23. vhdl-null.c — This file contains the custom VHDL version of the channel clock filters for Spectrum. The channel clock filters implement the Chandy-Misra synchronization method with null messages. The VHDL kernel requires a modification to the time filter to 'flush' pending transactions (using the **update_curr_value** function) from the driver whenever the time is advanced.

C.3 Changing Build-Generated Programs

After the BUILD phase generates the C program for a model, it must be changed to initialize the special data structures for the distributed kernel. The routines generated by BUILD for each VHDL process do not need to be changed for the distributed kernel. However, the **sim_initialize** routine must be changed.

The changes listed here will need to be inserted into the **sim_initialize** routine where existing code performs a similar function. The changes can be summarized as:

1. Concatenate the five files produced by build into a single file. The order should be the same as the file are listed in the file **sim100.c**, namely: **sim04.c**, **sim01.c**, **sim02.c**, **sim03.c**, **sim05.c**. This eliminates the need for **sim100.c**.

2. There are two sets of variables constructed by BUILD, which aren't used by the distributed simulator. There are two variables for every signal in the model; one

variable is **s_XX_wait_sens_proc_num**, and the other is **s_XX_wait_sens_proc_paused**. The **XX** will be the signal number used in the VIA file. All references to these variables and their declarations can be deleted.

3. Any signals that do not have a VHDL statement or process modifying them will not have a driver structure declared by the BUILD phase. The declarations for the driver pointers, and code for linking the driver structure to the signal structure need to be added. For example, if two signals (**s46**, and **a47**) do not have VHDL processes associated with them:

```
Driver *d46, *d47;

d46 = Newdrv();
d46->sig_ptr = s46;
s46->drv_ptr = d46;

d47 = Newdrv();
d47->sig_ptr = s47;
s47->drv_ptr = d47;
```

4. The **signumber** field in the signal structure needs to be initialized. This is best done where the **sig_array** is initialize. For example,

```
sig_array[0] = s58;
s58->signumber = 0;
sig_array[1] = s45;
s45->signumber = 1;
```

   The **signumber** contains the index of the signal in the **sig_array**.

5. Declare and initialize the **ps0** and **pd0** variables. These correspond to the sensitivity list for process 0, and the list of signals driven by process 0. The sensitivity list is a list of signal pointers, while the driven list is a list of pointers to drivers.

6. Create an entry in the process table (**pt**) for the input-vector manager process. Conveniently, the BUILD program leaves **pt[0]** unused, so the input-vector manager becomes process 0. For example,

```
pt[0].proc_ptr = NULL;  /* these two are always NULL */
pr[0].check_wait_cond = NULL;
pt[0].dlist_cnt = 4;    /* Number of signals driven by process 0 */
pt[0].dlist = pd0;
pt[0].slist = ps0;
ps0[0] = NULL;  /* No signals in sensitivity list */
pd0[0] = d46;   /* Signals driven by process 0    */
pd0[1] = d47;
pd0[2] = d48;
pd0[3] = d49;
s46->process_id = 0;
s47->process_id = 0;
s48->process_id = 0;
s49->process_id = 0;
```

This example illustrates a model with 4 signals (**s46**, **s47**, **s48**, and **s49**) handled by the input-vector manager.

7. For the remaining process table entries, the **dlist_cnt** field must be initialized to the number of signals driven by the process.

8. The **process_id** field in the signal structure must be set to the index (in the process table) of the process which drives the signal.

Appendix D. *Summary Paper*

D.1 Introduction

The Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) program was started in 1983 to standardize the tools used in Air Force applications to design, test and document large-scale digital electronic systems. In previous research, AFIT has been developing a set of VHDL tools, including an analyzer and simulator for Unix machines (9). These tools implement a subset of the complete VHDL language. VHDL allows the operation of a circuit to be simulated and tested before construction. This will reduce the time taken to test the circuit. As circuits have grown larger, the increased simulation time has reduced the benefits obtained by using VHDL. Therefore, we have explored using distributed simulation techniques to simulate VHDL models. Distributed simulation is a family of techniques for simulating a system by using separate processors communicating via message passing. Distributed simulation makes good advantage of the concurrency inherent in a VHDL circuit model. This work has explored the use of the Chandy-Misra null message algorithm to control VHDL simulation.

D.2 The Chandy-Misra Algorithm

The Chandy-Misra algorithm is used to synchronize the independent logical processes in a distributed simulation. The Chandy-Misra algorithm is a conservative simulation algorithm, since it constrains the actions of a logical process to prevent the need for rollback. Each logical process has an independent clock. Processes communicate via message-passing, with each message containing an event to be simulated and the simulation time of the sending process. Each logical process maintains an input-channel clock for each logical process from which it is capable of receiving a message. The input-channel clock is the time-stamp of the last message received over the channel. When a logical process is ready to simulate an event, it cannot simulate beyond the minimum of its input-channel times. If the logical process were to simulate past the minimum of its input-channel clocks, it would be possible for a message to arrive along the arc with a time stamp between the minimum and the current clock. This message is in the process' past, so the process must

roll back. Since the goal of Chandy-Misra is to prevent the need for rollback, the algorithm prevents this from occurring by blocking the logical process until he next event to be simulated is later than the minimum of the input-channel clocks.

Even with the constraints on advancing the simulation time, the simulation is still open to deadlock. Deadlock may occur when a logical process sends an event to one of several output processes. The input-channel time of the recipient is updated, but the remaining processes are still waiting for an event in order to update their input-channel time. Null messages are introduced into the simulation to avoid deadlock in this situation. Whenever a process sends an event to one of several output processes, the remaining processes are sent null messages to update their input-channel times. This will allow these processes to process events and proceed with the simulation.

D.3 The VHDL Language

VHDL contains concurrent statements that can model the concurrent activity within a circuit. These statements serve to model the signals contained within the VHDL model. This work has centered on two of the main concurrent statements of VHDL: the *process statement*, and the *concurrent signal assignment statement.* A process statement contains code that is executed sequentially to compute the future value of a signal. A concurrent signal assignment statement modifies a signal at some future simulated time, and can serve as a shorter form of a process block for simple expressions. When the VHDL model is simulated, both process statements and concurrent signal assignment statements are represented by *VHDL processes.* A VHDL process has a sensitivity list associated with it. A process is executed in response to a change in the value of a signal in its sensitivity list. The execution of a process results in new values for signals being computed for future simulation times. When the simulation clock advances, the changes in signal values trigger the execution of a different set of processes.

D.4 Basic Approach

In this work, each VHDL process is mapped into a single logical process within the simulation. A logical process is an abstraction that exists solely for the duration of a simu-

```
Algorithm D.1
while simulation is not done do
    wait for event from process
    if wait condition is satisfied then
        compute new output value
        if value has changed then
            send new value to dependent processes
        end if
    end if
end while
```

Figure D.1. Logical Process Algorithm

lation, and consists of a VHDL process combined with the Chandy-Misra synchronization and simulation control algorithm. Each logical process is responsible for maintaining the value of one or more signals. Whenever one of the signals in the sensitivity list changes, the logical process must update the values of its signals appropriately. If value of a signal changes as a result of this computation, it must be sent to any other process which lists the signal in its sensitivity list. This notifies the other logical process of the signal change. Therefore, each logical process executes the algorithm in Figure D.1. The events sent between process correspond to the changes in signal values.

This work makes use of the Spectrum distributed simulation testbed developed at the University of Virginia (26). Spectrum provides a standard environment for implementing distributed simulation applications, and can be customized through the use of filters which modify the simulation control method in use. For this VHDL simulation, Spectrum is used with a slightly customized version of the Chandy-Misra null message protocol filters. These filters generate and process all of the null messages required to maintain the correctness of the simulation. Spectrum provides a standard, portable environment for comparing different distributed simulation applications.

Dr. Paul Reynolds at the University of Virginia has also been developing some characteristics of distributed simulation applications and the effects that the characteristics have on performance (25). The performance of different simulation control algorithms (such as Chandy-Misra variants or Jefferson's Time Warp system (14)) depends, in part, on the

attributes of the application. Some of the attributes developed by Reynolds are listed in Table D.1, along with how they apply to VHDL simulation.

The first characteristic is *determinism*. An application may show varying degrees of determinism ranging from deterministic to nondeterministic. A deterministic simulation will give the same results when executed several times on a sequential processor. *Queuing* indicates a paradigm where the time for processing an event at a logical process is dependent on the presence of other events. Queuing occurs in network simulations where messages are queued for processing; logic simulations are one type of simulation where this doesn't occur. *Processing delays* occur if an event emitted by an logical process has a simulation time greater than the arriving event. *Causality* indicates the degree to which every event arriving at a logical process leads to an identifiable subsequent event leaving the logical process. Causality is one of the more critical characteristics that a simulation control mechanism has to contend with. *Production* occurs when an event can be created by a logical process, while *consumption* occurs when a logical process may take in an event without generating an output event. Another characteristic is how objects *change state* in a simulation: how many objects change and with what frequency. *Balance* is a measure of the uniformity of processing requirements across the logical processes. Balance has implications for ensuring that all processors are kept busy in a distributed simulation. The *level of activity* in a simulation is based on the number of logical processes that are busy at any instant of time. A low level of activity may signal little opportunity for parallelism in a simulation. *Connectivity* measures how much events in one logical process can directly affect events in another. This measure includes the direction of information flow in a simulation, and the topology of the inter-process communications network.

Most of Reynold's characteristics depend on the actual VHDL circuit under simulation, so it is hard to draw general conclusions about VHDL simulation. A VHDL simulation is consumptive since an input-event may not generate an output-event if the value of the output signal hasn't changed. The simulation control mechanism, in this case—Spectrum, must ensure that consumption of events doesn't cause deadlock. Since so many of these characteristics depend on the circuit, it seems that the performance of any one distributed simulation kernel will depend on the circuit being simulated.

Table D.1. Characteristics of VHDL Simulation

Reynold's Characteristic	VHDL Property
Determinism	Model-Dependent
Queueing	Absent
Processing Delays	Present
Causality	Consumptive
Balance	Model-Dependent
Activity Level	Model-Dependent
Connectivity	Model-Dependent

The Spectrum system and distributed VHDL kernel have been implemented and tested on the Intel iPSC/2 Hypercube system at AFIT. The iPSC/2 has eight nodes, each containing an 80386 processor and 8 megabytes of RAM. The eight nodes are connected using a high-speed cube-connected network. The cube is also connected to a host system which controls access. Each logical process in the VHDL simulation executes on one processors as a task controlled by the iPSC/2 node operating system. The assignment of logical processes to nodes is made when the simulation is loaded onto the cube, and is arbitrary.

D.5 Performance

This section describes the performance measurement technique used to evaluate the distributed VHDL kernel and the results of the evaluation.

D.5.1 Measurement Technique There are two possible methods to measure the execution time of an application executing on the iPSC/2. The host (cube manager) can measure the time taken for the node processes to complete, or the node processes can time themselves. The timer on the host provides the time to the nearest second, which is too coarse for evaluating VHDL simulation. Each node has a free-running timer with millisecond resolution, which allows a node process to sample the elapsed time since the process started. When tasking is used on a node, this timer doesn't provide the individual times for all of the tasks. The execution of all of the tasks on the node is interleaved during the

Table D.2. Performance of Test Cases

Test Case	Number of Nodes 2	4	8
FREEADD	53.26	53.11	53.02
COUNT	27.85	25.92	30.25
LOOK	46.80	46.70	46.80

time interval reported. Therefore, it is impossible to separate the times for the individual tasks.

These limitations require using only the maximum time reported by a logical process as the time for a simulation. This is because the simulation cannot be considered complete until the last logical process completes.

D.5.2 Test Cases Three test cases were executed on various numbers of processors to develop the performance profile of this distributed kernel. The test case were: an 8-bit ripple carry adder (FREEADD), a 3-bit counter (COUNT), and a 4-bit carry generate adder (LOOK). These circuits test both combinational and sequential logic, and demonstrate many of the features supported by this kernel. The FREEADD case consists of 33 logical processes, COUNT consists of 7 logical processes, and LOOK consists of 37 logical processes. The performance of the test cases is summarized in Table D 2. Note: the COUNT simulation can use only 7 of the 8 available processors.

These test cases shows almost identical performance on different numbers of processors. Therefore, the speed-up obtained is approximately 1. This seems to indicate a mismatch between the kernel design and the test cases. All of the test cases are relatively low-level models, with very little computation to do for each event received by the logical processes. Therefore, the simulation spends most of its time waiting for an event to arrive.

In addition, the kernel does not take advantage of logical processes that share the same processing node. When the FREEADD simulation is executed on 2 processors, one node has 16 logical processes assigned and the other has 17 logical processes. Each logical process is completely independent of the others and maintains an independent simulation

clock. Therefore, null messages are exchanged between logical processes on the same node for synchronization as required by the Chandy-Misra algorithm within Spectrum. These null messages are a major source of overhead in the simulation, since they could be eliminated if all logical processes assigned to a node shared a single simulation clock.

Even with the kernel problems above, this kernel design may still be effective for simulations that are composed of a small number of computationally intense logical processes. Unfortunately, this hypothesis is untested since the subset nature of the AFIT VHDL tools make it difficult to use large off-the-shelf VHDL models.

D.6 Conclusions

Distributed simulation can be used very naturally for simulating VHDL circuits. Since VHDL has many concurrent statements and other features, distributed simulation amounts to concurrent execution of a concurrent language. The Spectrum distributed simulation testbed allows simulation applications (such as VHDL) to be developed independent of the underlying synchronization method. The performance of a VHDL simulation kernel seems to depend on the characteristics of the model under test. Different kernels (synchronization methods) will have different performance with different models.

Bibliography

1. Berk, Kevin J. *Impact of IEEE Standard 1076 on VHDL.* MS thesis, AFIT/GCE/ENG/88D-1. Air Force Institute of Technology, December 1988 (AD-A202739).
2. Bryant, Randal E. "Simulation on a Distributed System." In *Proceedings of the 1st Int'l Conference on Distributed Computing Systems*, pages 544–552, October 1979.
3. Chandy, K. M. and J. Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, *24*(11):198–206 (1981).
4. Chandy, K. Mani and Jayadev Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, *SE-5*(5):440–452 (1979).
5. Chandy, K. Mani, et al. "Distributed Deadlock Detection," *ACM Transactions on Computer Systems*, *1*(2):144–156 (May 1983).
6. Chandy, K.M. and J. Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, *24*(11):198–206 (April 1981).
7. Chawla, Praveen. *Functional Digital System Simulation in a Multiprocessor Environment Using the Virtual Time Algorithm.* MS thesis, University of Cincinnati, 1988.
8. Chung, Moon Jung and Yunmo Chung. "Data Parallel Simulation using Time-Warp on the Connection Machine." In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, 1989.
9. DeGroat, Joseph W., et al. "The AFIT VHDL Environment." In *Proceedings of the IEEE 1988 Frontiers in Education Conference*, pages 324–330, 1988.
10. Dewey, Allen and Anthony Gadient. "VHDL Motivation," *IEEE Design and Test*, pages 12–16 (April 1986).
11. Fujimoto, Richard M. "Performance Measurements of Distributed Simulation Strategies." In *Distributed Simulation 1988*, La Jolla CA: SCS, 1988.
12. Hoare, C.A.R. *Communicating Sequential Processes.* Englewood Cliffs NJ: Prentice-Hall, 1985.
13. The Institute of Electrical and Electronics Engineers. *IEEE Standard VHDL Language Reference Manual*, 1987.
14. Jefferson, David R. "Virtual Time," *ACM Transactions on Programming Languages and Systems*, 7(3):404–425 (1985).
15. Kaudel, Fred J. "A Literature Survey on Distributed Discrete Event Simulation," *Simuletter*, *18*(2):11–21 (June 1987).
16. Lipsett, Roger, et al. "VHDL — The Language," *IEEE Design and Test* (1986).
17. Luckham, David, et al. "The Semantics of Timing Constructs in Hardware Description Languages." In *Proceedings of the IEEE 1986 International Conference in Computer Design*, 1986.

18. Mannix, David Louis. *Distributed Discrete-Event Simulation Using Variants of the Chandy-Misra Algorithm on the Intel Hypercube*. MS thesis, AFIT/GCS/ENG/88D-14. Air Force Institute of Technology, December 1988 (AD-A202849).

19. Matechik, Steven M. *Graphical VHDL User Interface*. MS thesis, AFIT/GE/ENG/88D-25. Air Force Institute of Technology, December 1988 (AD-A202619).

20. Misra, Jayadev. "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, *18*(1):39–65 (March 1986).

21. Pompilio, Douglas G. *Hardware Modelling with VHDL Simulation*. MS thesis, AFIT/GCS/ENG/88D-15. Air Force Institute of Technology, December 1988 (AD-A206356).

22. Preiss, Bruno R., et al. "A unified modeling methodology for performance evaluation of distributed discrete event simulation mechanisms." In *Proceedings of the 1988 Winter Simulation Conference*, 1988.

23. Quinn, Michael J. *Designing Efficient Algorithms for Parallel Computers*. Mc Graw-Hill, Inc., 1987.

24. Reed, Daniel A. and Allen D. Malony. "Parallel Discrete Event Simulation: The Chandy-Misra Approach." In *Distributed Simulation 1988*, La Jolla CA: SCS, 1988.

25. Reynolds, Jr., Paul F. *A Comparative Analysis of Parallel Simulation Protocols*. Technical Report, Virginia Institute for Parallel Computation, 1989.

26. Reynolds, Jr., Paul F. and Phillip M. Dickens. *Spectrum: A Parallel Simulation Testbed*. Technical Report, Virginia Institute for Parallel Computation, 1989.

27. Soule, Larry and Tom Blank. "Parallel Logic Simulation on General Purpose Machines." In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, 1988.

28. Soule, Larry and Anoop Gupta. "Characterization of Parelism and Deadlocks in Distributed Digital Logic Simulation." In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, 1989.

29. Stone, Harold S. *High-Performance Computer Architecture*. Addison Wesley, 1987.

30. Su, Wen-King and Charles L. Seitz. "Variants of the Chandy-Misra-Bryant distributed discrete-event simulation algorithm." In *Distributed Simulation 1989*, 1989.

Vita

Captain Michael Proicou [redacted] attended Purdue University in West Lafayette, Indiana. While at Purdue, he was a member of the Air Force ROTC detachment. He graduated with a degree in Computer Science and Mathematics and was commissioned a Second Lieutenant on May 17, 1985. After assignment to Headquarters Air Force Logistics Command, Wright-Patterson AFB, Ohio, Lieutenant Proicou served as a logistics policy analyst and managed computer systems within the Materiel Management Deputate. In May 1988, he entered the graduate computer sciences program at the Air Force Institute of Technology.

[redacted]

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/89D-14	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology (AU) Wright-Patterson AFB, OH 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Wright Res. Dev. Center 8b. OFFICE SYMBOL (If applicable) WRDC/AADE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Wright Research Development Center Wright-Patterson AFB, OH 45433	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) A DISTRIBUTED KERNEL FOR SIMULATION OF THE VHSIC HARDWARE DESCRIPTION LANGUAGE	
12. PERSONAL AUTHOR(S) Michael C. Proicou, Captain, USAF	
13a. TYPE OF REPORT MS Thesis 13b. TIME COVERED FROM _____ TO _____ 14. DATE OF REPORT (Year, Month, Day) December 1989 15. PAGE COUNT 128	
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Computerized Simulation; Distributed Simulation
12	05		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Advisor: Bill Hodges, Capt, USAF

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Bill Hodges, Capt, USAF	22b. TELEPHONE (Include Area Code) (513) 255-5533 22c. OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 *Previous editions are obsolete.*

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Abstract

This thesis develops a technique for simulating large-scale circuit models, written in the Very High Speed Integrated Circuit Hardware Description Language (VHDL), on a distributed computer composed of many individual processing units. The distributed system consists of a scalable kernel which can support a large simulation composed of many concurrent VHDL processes. The kernel provides model-independent support functions that handle signal propagation and process activation in a distributed environment. The synchronization between individual logical processes in the kernel is handled using the Chandy-Misra null message algorithm.

The distributed kernel has been implemented using the Spectrum distributed simulation testbed and runs on the eight-processor Intel iPSC/2 Hypercube at AFIT. Several test circuits have been simulated, including ripple-carry and carry look-ahead adders, and a counter. These cases cover combinational and sequential logic.